MathBook XML Author's Guide

MathBook XML Author's Guide

Robert A. Beezer
University of Puget Sound

DRAFT June 19, 2017 DRAFT

Contents

Chapter 1

Start Here

Welcome to the Author's Guide for MathBook XML. You are likely eager to get started, but familiarizing yourself with this chapter should save you a lot of time in the long run. We will try to keep it short and at the end of early chapters we will guide you on where to go next. Not everything we say here will make sense on your first reading, so come back after your first few trial runs.

1.1 Philosophy

MathBook XML is a **markup language**, which means that you explicitly specify the logical parts of your document and not how these parts should be displayed. This is very liberating for an author, since it frees you to concentrate on capturing your ideas to share with others, leaving the construction of the visual presentation to the software. As an example, you might specify the content of the title of a chapter to be `Further Experiments`, but you will not be concerned if a 36 point sans-serif font in black will be used for this title in the print version of your book, or a CSS class specifying 18 pixel height in blue is used for a title in an online web version of your book. You can just trust that a reasonable choice has been made for displaying a title of a chapter in a way that a reader will recognize it as a name for a chapter. (And if all that talk of fonts was unfamiliar, all the more reason to trust the design to software.)

You are also freed from the technical details of presenting your ideas in the plethora of new formats available as a consequence of the advances in computers (including tablets and smartphones) and networks (global and wireless). Your output "just works" and the software keeps up with technical advances and the introduction of new formats, while you concentrate on the content of your book (or article, or report, or proposal, or ...).

If you have never used a markup language, it can be unfamiliar at first. Even if you have used a markup language before (such as HTML or basic LaTeX) you will need to make a few adjustments. Most word-processors are WYSIWYG ("what you see is what you get"). That approach is likely very helpful if you are designing the front page of a newspaper, but not if you are writing about the life-cycle of a salamander. In the old days, programs like `troff` and its predecessor, `RUNOFF` (1964), implemented simple markup languages to allow early computers to do limited text-formatting. Sometimes the old ways are the best ways.

MathBook XML is what is called an **XML application** or an **XML vocabulary** (I prefer the latter). Authoring in XML might seem cumbersome at first, but you will eventually appreciate the long-run economies, so keep an open mind. And if you are already familiar with XML, realize we have been very careful to design this vocabulary with human authors foremost in our mind.

Principles The creation, design, development and maintenance of MathBook XML (MBX here) is guided by the following list of principles. They may not be fully understood on a first reading, but should be useful as you become more familiar with authoring texts with MathBook XML and should amplify some of the previous discussion.

List 1.1.1 (MathBook XML Principles).

1. MBX is a markup language that captures the structure of textbooks and research papers in the mathematical sciences.
2. MBX is human-readable and human-writable.
3. MBX documents serve as a single source which can be easily converted to multiple other formats, current and future.
4. MBX respects the good design practices which have been developed over the past centuries.
5. MBX makes it easy for authors to implement features which are both common and reasonable.
6. MBX supports online documents which make use of the full capabilities of the Web.
7. MBX output is styled by selecting from a list of available templates, relieving the author of the burden involved in micromanaging the output format.
8. MBX is free: the software is available at no cost, with an open license. The use of MBX does not impose any constraints on documents prepared with the system.
9. MBX is not a closed system: documents can be converted to LaTeX and then developed using standard LaTeX tools.

1.2 Formatting Your Source

There are a lot of details related to how you prepare your **source**: the actual files that you, and you alone, will create. At least skim through the following, come back here often, and also consult ⟨⟨exhaustive-chapter-on-source-files⟩⟩.

File Format Your source should be plain **ASCII files** which you will create with a text editor. In other words, do not create your source with Word, LibreOffice, WordPerfect, AbiWord, Pages or similar programs. Popular text editors include vi, emacs, Notepad, Notepad++, Atom, TextWrangler, and BBEdit. I have had a very good experience with Sublime Text, which is cross-platform (Windows, OS X, Linux), and can be used for free, though it has a very liberal license and is well worth the cost. Sometimes these editors are known as a **programmer's editor** (though we will be doing no programming). Support for writing HTML sometimes translates directly to good support for XML.

There are **XML editors**, which I have generally found too complex for authoring in MathBook XML. They do have some advantages and XML Copy Editor is one that I have found that is possibly useful.

Learn to Use Your Editor Because XML requires a closing tag for every opening tag, it feels like a lot of typing. Your editor should know what tag to close next and there should be a simple command to do that. Discover this first and consider switching editors if it is not available. For me, in Sublime Text on Linux, I just press `Alt-Period` and get a closing tag. Not only is this quick and easy, I often recognize that I am not getting the tag I expected since I forgot to close one earlier. This one shortcut can pretty much cut your authoring overhead in half.

If your editor can predict your opening tag, all the better. Sublime Text recognizes that I already have a `<section>` elsewhere, so when I start my second section, I very quickly (and automatically) get a short list of choices as I type, with the one I want at the top of the list, or close to it.

Invest a little time early on to learn, and configure, your editor and you can be even more efficient about capturing your ideas with a minimum of overhead and interference.

Revision Control If you are writing a book, or if you are collaborating with co-authors, then you owe it to yourself and your co-authors to learn how to use revision control, which works well with MathBook XML. The hands-down favorite is `git` which has a steep learning curve, and so is beyond the scope of this guide. But see ⟨⟨topic-git-coexistence⟩⟩ which has hints on how to best use `git` together with a MathBook XML project and look for Beezer and Farmer's *Git For Authors*.

Whitespace The term **whitespace** refers to characters you type but typically do not see. For us they are **space**, **non-breaking space**, **tab** and **newline** (also known as a "carriage return" and/or "line feed"). Unlike some other markup languages, MathBook XML *does not ever use whitespace* to convey formatting information.

In some parts of a MathBook XML document, every single whitespace character is important and will be transmitted to your output, such as in the `<input>` and `<output>` portions of a `<sage>` element. Since Sage code mostly follows Python syntax, indentation is important and leading spaces must be preserved. But you can indent all of your code to match your XML indentation and the entire `<input>` (or `<output>`) content will be uniformly shifted left to the margin in your final output.

In other parts of a MathBook XML document, every single whitespace character is ignored, and you have the freedom to use indentation and blank lines to help you understand the logical structure of your document. An example is that you can add as much whitespace as you like between the paragraphs of a section, such as a preceding blank line and indentation, and none of it will affect your output in any way.

Never use tabs, they can only cause problems. You may be able to set your editor to translate the tab key to a certain number of spaces, or to translate tabs to spaces when you save a file (and these behaviors are useful). I have Sublime Text configured to show me every single space as a small faint dot, since I like to be certain I have no stray whitespace *anywhere*.

Structure of your Source XML is hierarchical, like a family tree. Books contain chapters, chapters contain sections, sections contain paragraphs, etc. I like to reflect each new level of containment by consistently indenting four spaces (you might prefer two spaces). A good editor will visually respect this indentation, and help you with maintaining the right indentation with each new line, so much so, that you will forget how much assistance it is providing.

Develop a style and stick with it. I put titles on a new line (indented) after I create a new chapter or section, some people like them on the same line, immediately adjacent. I put a single blank line before each new paragraph, but not after the last. And so on. The choice is yours, but consistency will pay off when you inevitably come back to edit something. You have put a lot of work and effort into your source. You will be rewarded with fewer problems if you keep it neat and tidy, and you will also get very clean output.

1.3 Where Next?

We will end each of the early chapters with suggestions on where to read next.

If you are impatient (sometimes a good quality!), then Chapter 2 should be next, where we construct and process a short example and then expand it slightly with several additions.

If you would like a general, high-level overview of features skip ahead to Chapter 3.

Chapter 2

A Careful, Quick, Minimal Example

We are going to walk you through a simple example carefully, with some explanation along the way. There are two steps. First we will author your source and then after some setup, we will process your source file into output.

2.1 Authoring

Having read already about text editors in Section 1.2, fire up your text editor and begin a text file that you will name `quickstart.xml`.

Though it is optional, it will be good practice to always make the first line of your file the following. This will identify your source file as an XML file, but the exact particulars are not important now.

```
<?xml version="1.0" encoding="UTF-8"?>
```

Every MathBook XML document is enclosed inside `<mathbook>` tags and we will be writing an `<article>`. So extend your source to the following.

```
<?xml version="1.0" encoding="UTF-8"?>
<mathbook>
    <article>
    </article>
</mathbook>
```

Notice how every opening tag (no slash) has a paired closing tag (with a slash, plus the identical name). We would like to our article to have some content, so we will add a short paragraph with a single sentence. So finish your file off as follows and save it.

```
<?xml version="1.0" encoding="UTF-8"?>
<mathbook>
    <article>
        <p>This is a short sentence.</p>
    </article>
</mathbook>
```

That's it. You now have a complete, properly-formed MathBook XML document. You are ready to organize the two tools necessary to process your source into different formats.

2.2 Setup

There are two components to processing your document, the MathBook XML stylesheets and the `xsltproc` program. We work at the **command-line** inside of a **terminal** or **console**. If you do not know what this

is, it will seem very primitive at first. Sometimes the old ways are the best ways. This will be called a "Command Prompt" in Windows or a "Terminal" on a Mac. In Linux it may be known as a "console" or a "shell". A tutorial, which is Linux-specific, can be found at Ryan's Tutorials and certainly others exist.

The operating system on a Mac is built on Unix, which is very similar to Linux, so most of the directions here will be little changed between the two. Procedures can be very different in Windows. One alternative is SageMathCloud which provides a full Linux computer for free in your web browser, so that may be an excellent place for initial experiments.

Step 1: MathBook XML You need to obtain the MathBook XML stylesheets, which are the main part of MathBook XML. Since you are reading this, it may be possible that you have this already. You can use `git` to **clone** the MathBook XML from the GitHub repository, and then be sure to checkout the `dev` branch to have the latest version. This is a good way to go, since you can then easily "pull" updates to the stylesheets.

If you do not want to install `git` you can get an archive of all the files to unpack on your system. Just remember that you will need to redo this each time you want to get updates.

See the website at `mathbook.pugetsound.edu` for details and commands for this step.

Step 2: `xsltproc` This is the command-line program which takes your document and a MathBook XML stylesheet to together produce output. On Linux or a Mac you probably already have it installed as part of system software. On Windows it is not so simple.

In either case see the website for details abut verifying you have this, or how to install it.

2.3 Processing

At a command prompt in your terminal or console adjust the pathnames for the two files and execute:

```
$ xsltproc /path/to/mathbook/xsl/mathbook-html.xsl /path/to/quickstart.xml
```

In the current working directory you should now find the file `article-1.html` which you can view in a web browser. (You will want an internet connection since various parts of the page come from the network. Someday we will create output for the offline situation.) It will look very plain, but you should be able to read the sentence.

Now, try the following, again with adjusted paths:

```
$ xsltproc /path/to/mathbook/xsl/mathbook-latex.xsl /path/to/quickstart.xml
```

In the current working directory you should now find the file `article-1.tex` which you can process with `pdflatex` or `xelatex` at the command line as below. If you do not have LaTeX installed on your system, you could process this file within a variety of online services, and SageMathCloud would be an obvious choice.

```
$ pdflatex article-1.tex
```

In the current working directory you should now find the file `article-1.pdf` which you can view or print with standard PDF viewing software. You could even send it to a print-on-demand service to get nice hardback books, though I suspect sales will not be great.

That's it. You now know all the basics of authoring with MathBook XML, since you have produced two radically different output formats with identical content from the exact same structured input, via two different command lines. Everything you need to author a complete article or textbook, and produce it in many different formats, is just an extension or variation on what you just did. Let us look at a few simple extensions right away before being more methodical.

2.4 Extending the Minimal Example

We will not keep reproducing the entire example but instead suggest a series of modifications. After each edit, process the file again to make sure your syntax is correct (before you get too far along) and to see the changes.

The generic name of the resulting files is pretty bland, and it would be nice to have a title for our article. We will add an **attribute** to the `<article>` tag, specifically `@xml:id`, which is a very important part of MathBook XML and used frequently. For now, it will be used to generate the names of the output files. (The "at" symbol is a way of reminding you that it is an attribute, it is not part of what you author.)

So make the following modifications:

```
<article xml:id="quick">
    <title>My First Small Example</title>

    <p>This is a short sentence.</p>
```

Your outputs should now have a title, and more importantly, the filenames will be `quick.html` and `quick.tex`. Of course, you might like your outputs to have similar names to your input, but you see that this is not necessary.

Let us give our article a bit of structure. We will have an introduction and two sections with their own titles. So replace the one-sentence paragraph by the following, all following the article title and contained within the `<article>` tags. Remember, *do not* include any newlines (carriage returns, line feeds, hard wrap) in the longer lines. (We have to format things differently so you can see exactly what is happening.)

```
<introduction>
    <p>Let's get started.</p>
</introduction>

<section xml:id="section-short">
    <title>Beginnings</title>

    <p>This is a short sentence.</p>
</section>

<section xml:id="section-multiple-paragraph">
    <title>Endings</title>

    <p>This is a longer sentence that is followed by another sentence.
    Two sentences, and a second paragraph to follow.</p>

    <p>One more paragraph.</p>
</section>
```

The LaTeX/PDF output will be a bit odd looking since every paragraph is so short, but all the content should be there. Notice that the HTML output now has a table of contents to the left and active navigation buttons. Also the two sections are in their own files and the URLs have been constructed from the supplied values of the `@xml:id` attribute.

One last experiment—let us add some mathematics. We use XML tags, `<m>` for "inline" mathematics and `<me>` for a "math equation" which will be rendered with a bit of vertical separation and centered from left to right. We use LaTeX syntax for mathematics, which has been the standard for working mathematicians for decades. For electronic presentation, we rely on the excellent MathJax project which basically supports all the syntax of the amsmath package from the American Mathematical Society. Add the following sentence to any of the paragraphs of your article.

```
If the two sides of a right triangle have lengths <m>a</m>
and <m>b</m> and the hypotenuse has length <m>c</m>,
then the equation <me>a^2 + b^2 = c^2</me> will always hold.
```

So your final source file might look like the following. You now have many of the basic skills you would need to write an entire research article in mathematics, and should be in a position to learn the remainder easily and quickly.

```
<mathbook>
    <article xml:id="quick">
        <title>My First Small Example</title>

        <introduction>
            <p>Let's get started.</p>
        </introduction>

        <section xml:id="section-short">
            <title>Beginnings</title>

            <p>This is a short sentence.</p>
        </section>

        <section xml:id="section-multiple-paragraph">
            <title>Endings</title>

            <p>This is a longer sentence that is followed by another sentence.
            Two sentences, and a second paragraph to follow.</p>

            <p>One more paragraph.  If the two sides of a right triangle have
            lengths <m>a</m> and <m>b</m> and the hypotenuse has length <m>c</m>,
            then the equation <me>a^2 + b^2 = c^2</me> will always hold.</p>
        </section>
    </article>
</mathbook>
```

2.5 Where Next?

Next chapter has more detail.

Chapter 3

Overview of Features

This chapter is a high-level view of the important concepts, features and design decisions that go into the creation of MathBook XML. For careful exact descriptions of details, we will direct you to one of the many sections in the (*) Topics chapter. So this chapter should make you aware of what is possible and expand on the philosophy described earlier in Section 1.1, while also giving you examples of many basic constructions you can use to get started quickly.

3.1 Structure

A MathBook XML document is a nested sequence of **structural subdivisions**. For a book, these would go `<part>`, `<chapter>`, `<section>`, `<subsection>`, and `<subsubsection>`. Using `<part>` is optional (and not yet fully implemented), but a book must always use `<chapter>` (or else it is not a book!). No skipping over subdivisions. For example, you cannot subdivide a `<section>` directly into several `<subsubsection>`s without an intervening `<subsection>`.

An `<article>` starts subdivisions from `<section>`, though it may choose to have no subdivisions at all. `<paragraphs>` are exceptional. They lack a full set of features, but can be used to subdivide anything, in books or in articles, though they are always terminal since you cannot subdivide them further. You will have noticed that we prefer the generic term **subdivision** (rather than "section") since a `<section>` is a very particular subdivision.

A subdivision may be unstructured, in which case you fill it with paragraphs and lists and figures and theorems and so on. But if you choose to structure a subdivision it must look like an optional `<introduction>`, followed by multiple subdivisions of the next finer granularity, with an optional `<conclusion>`.

Every subdivision tag can carry an `@xml:id` attribute, and it is a good practice to (a) provide one, (b) use a very short list of words describing the content, and (c) adopt a consistent pattern of your choosing. Do not use numbers, you may later regret it. These are optional, and with practice you will learn how best to use them. See Section 3.3 just below for more on this.

The `<exercises>` and `<references>` tags are special subdivisions, see Section 6.1 and Section 6.3.

This explanation is expanded and reiterated at Section 6.5 and is worth reading earlier rather than later.

3.2 Paragraphs

Once you have subdivisions, what do you put into them? Most likely, paragraphs. We use long, exact names for tags that are used infrequently, like `<subsubsection>`. But for frequently used elements, we use abbreviated tags, often identical to names used in HTML. So a paragraph is delimited by simply the `<p>` tag.

Lots of things can happen in paragraphs, some things can *only* happen in a paragraph, and some things are *banned* in paragraphs. Inside a paragraph, you can emphasize some text (`<em>`), you can quote some text (`<q>`), you can mark a phrase as being from another language (`<foreign>`), and much more. You can use special characters like an ampersand (&, `<ampersand />`) or an octothorpe, aka "hash tag" (#, `<hash />`). You must put a list inside a paragraph, and all mathematics (Section 3.5) will occur inside a paragraph. You

cannot put a <table> or a <figure> in a paragraph, and many other structured components are prohibited in paragraphs.

Paragraphs are also used as part of the structure of other parts of your document. For example, a <remark> could be composed of several <p>. As you get started with MathBook XML, remember that much of your actual writing will occur inside of a <p> and you will have a collection of tags you can use there to express your meaning to your readers.

3.3 Cross-References

Any element that you place a @xml:id on can become the target of a cross-reference. This could be a subdivision, a remark, a bibliographic entry, or a figure. So for example, suppose your source had <subsection xml:id="subsection-flowers"> and someplace else you wrote <xref ref="subsection-flowers" />. Then at the latter location you would get a reference to the Subsection that discusses flowers. In print this might just be the number for the subsection, but in various electronic output formats, these cross-references can be very powerful interactive ways to explore the content. And the mechanism is always the same, pair up an @xml:id on a target with a @ref on an <xref> cross-reference.

Since the value of an @xml:id is also used in a variety of ways, such as to construct some file names, some care should be taken in how you author them. We limit the possible characters to letters and numbers (a-z, A-Z, 0-9), with hyphens and underscores (-_) available as word-separators. Our advice is to stick to lowercase letters, though we are not yet aware of any problems with case-insensitivity. So in short, use **kebab-case** or **snake-case** for your @xml:id values.

For more, see Section 6.4 because cross-references have many features. But first, here are two features you do not want to miss. In the early stages of writing, you can author <xref provisional="subsection-flowers" /> to point to a subsection you are contemplating (but have not written yet) and you will get various polite reminders to get that straightened out eventually. Also there is an **autoname** feature that will automatically provide the generic name of the target, so you will get something like "Subsection 4.3.2" without ever typing the "Subsection" part. If you move the target, the generic name will adjust if necessary, and if you switch to to one of the supported languages, the generic name will switch language (see ⟨⟨topic-on-languages⟩⟩.

3.4 Titles

Subdivisions always need titles, you accomplish this with a <title> tag first thing. Almost everything that you can use in a paragraph can be used in a title, but a few constructions are banned, such as a displayed mathematical equation (for good reason). Try to avoid using footnotes in titles, even if we have tried to make them possible.

Lots of other structures admit titles. Experiment, or look at specific descriptions of the structure you are interested in. Titles are very integral to MathBook XML, much like cross-references. Titles migrate to the Table of Contents, get used in page headers for print output, can be used in lists (such as a List of Figures), and can be used as the text of a cross-reference, instead of a number. You might not be inclined to give a <remark> a title, but it would be good practice to do so. Your electronic outputs will be much more useful to your readers if you routinely title every structure that allows.

3.5 Mathematics

A key design decision is that mathematical symbols, expressions and equations are authored using LaTeX syntax. More precisely, we support the symbols and constructions provided by MathJax, which quite closely follows the amsmath package maintained by the American Mathematical Society. Neither you nor I want to write MathML by hand!

For **inline** mathematics, use the short <m> tag within a <p> (or within a <title> or <caption>). For example, <m>\alpha^2 + \beta^4</m> will do what you expect, in print and in electronic outputs. To get a single equation, centered, with some vertical separation before and after, use the <me> tag ("math equation")

in the same way within a <p>, but do not try using it within a <title>. For example, <me>\rho = \alpha^2 + \beta^4</me>. If you want your equation numbered, switch to the <men> tag (n = "numbered").

There is a way to incorporate your own (simple) custom LaTeX macros within mathematics (only). They will be effective in your print and electronic outputs, and can be employed in graphics languages like tikz and Asymptote. You can also author multi-line **display mathematics** using the <md> tag surrounding a sequence of <mrow> elements (or the <mdn> variant for numbered equations). We defer the details to Section 6.6.

3.6 Images

You can include an image via the <image> tag. You can use the @source attribute to provide a filename, likely prefixed by a relative path, that points at an image file. It is your responsibility to locate that file properly relative to your output, and that the file format is compatible. So, for example, suppose your source contained <image source="images/butterflies.jpg" />. Then you would want to have a directory named images below wherever you process your LaTeX output, or wherever you place your HTML output on a web server. The @width attribute can be used to control the size of the image. Specifying a percentage is easiest, such as width="60%".

You will probably want to wrap your image in a <figure> to have it centered, and to have some vertical separation above and below. Presence of a <caption> element will dictate whether or not the figure is numbered. See Section 3.10 for more. Note too, that the <sidebyside> tag provides some very flexible options for placing several images (Section 3.15) together.

If you wish to construct technical diagrams, with editable source, and perhaps including the use of LaTeX macros, MathBook XML provides support for graphics languages like PGF, TikZ, and Asymptote, in addition to using Sage code to describe a plot or image. In most cases output can be obtained as smoothly-scalable SVG images, in addition to other formats like PDF or PNG. Making all this happen is one of the more technical aspects of MathBook XML, so we save the discussion for one of the topics, ⟨⟨topic-image-languages⟩⟩.

3.7 Lists

Ordered lists (numbered), unordered lists (bullets) and description lists (defined terms) are all supported, and syntax generally follows HTML. Lists always live within a paragraph (<p>). Their structure is given by the <ol>, <ul>, <dl> tags (respectively). These can specify a variety of options for the labels via attributes, as described in Section 6.7. Because a paragraph has mixed-content, you want to place these list-initiating tags immediately after the character preceding it, with no intervening whitespace, and specifically not with an intervening newline.

Once inside a list, you may use as much whitespace as you choose between the list items. Typically you would indent one level, start each list item on its own line, and perhaps use blank lines between list items of a complicated list.

List items, for any of the three types, are delimited with the <li> tag. What is different from HTML is that the contents of a list item may be structured, with paragraphs (<p>) being the most likely. So to nest lists you begin a paragraph in a list item of the outer list, then begin the inner list within that paragraph. But a simple list item may be authored as mixed-content, much like a sentence elsewhere. For a description list, the list item should contain a <title>, which will become the text that is being described. A description list typically does not contain a nested list, nor is a description list contained within another list.

3.8 Exercises

An exercise can appear in the narrative as an **inline exercise** or in a special <exercises> subdivision and is then known as a **sectional exercise**. The syntax is the same in both cases, there are just some minor differences in their use and treatment. The <exercises> subdivision can be used at any level. In other words, it can be a peer of any other subdivision.

An <exercise> is a mandatory <statement> followed by possibly several optional <hint>, <answer> and/or <solution>. Conceptually, an <answer> is a short final result, while a <solution> provides details about the route to the answer. Each of these four components is structured further, with paragraph-like elements, and the exercise itself may have a <title>.

You need (and want) to have the hints, answers and solutions grouped with the statement, but there is a lot of flexibility on making these available at the location of the exercise, or in the back matter. See Section 6.8 for more.

An inline exercise typically gets a fully qualified unique number and is rendered similar to an <example> or a <remark>. A sectional exercise only gets a sequential number, though this can be overridden with the @number attribute if you want to maintain stable numbering in response to edits. (Be careful, once you override the sequential numbering, you probably need to manually specify every subsequent number, so save overrides for when your project matures.)

Within a run of sectional exercises a subgroup can be delimited as an <exercisegroup>, which allows an <introduction> and/or <conclusion> to explain some commonality. An <exercisegroup> should be rendered in some way that makes it clear to the reader that they are a group.

3.9 References

Like <exercises>, a <references> subdivision may go anywhere a more typical subdivision could go. This allows for things like a "Further Reading" list at the end of every chapter of a book. These are populated with <biblio> items that are individual bibliographic entries. Support is presently very minimal, but is planned to improve.

3.10 Figures and Tables

Figures and tables have outer structures, <figure> and <table> which can contain a <title> (for cross-references, such as in a list of figures) and a <caption>. The presence of a <caption> generates a number for the figure or table. These outer structures also generate some vertical separation between surrounding elements.

A <table> may only contain a <tabular>, which is the structure containing the rows and columns of a table. Details can be found in Section 6.10.

A <figure> is more liberal than a <table> and may contain a variety of items. Most probably, you will include an <image> (see Section 3.6).

3.11 Programs and Consoles

If you are writing about scientific or engineering topics, you may wish to include sample computer programs, or command-line sessions.

A <program> will be treated as verbatim text (see Section 3.13), subject to all the exceptions for exceptional characters. Indentation will be preserved, though an equal amount of leading whitespace will be stripped from every line, so as to keep the code shifted left as far as possible. So you can indent your code consistently along with your XML indentation. For this reason it is best to indent with spaces, and not tabs. A mix will almost surely end badly, and in some programming languages tabs are discouraged (e.g. Python).

The @language attribute may be used to get some degree of language-specific syntax highlighting. See Section 6.11 for details.

A <console> is a transcript of an interactive session in a terminal at a command-line. It is a sequence of the following elements, in this order, possibly repeated many times as a group: <prompt>, <input>, and <output>. The <output> is optional, and a <prompt> is only displayed when there is an immediately subsequent <input> (which could be empty). The content is treated as verbatim text (see Section 3.13), subject to all the exceptions for exceptional characters.

A <program> or <console> may be wrapped in a <listing>, which is analogous to a <figure> or <table>. A <listing> may contain a <title> for use in cross-references or lists. A <caption> will be displayed and will generate a number.

3.12 Special Characters

An advantage of XML syntax is that very few characters are reserved for the language's use, and thus very few characters need to be escaped. Of course, there is always the need to escape the escape character.

The escape character for XML is the ampersand, &. The other dangerous character is the left angle bracket, the "less than," <. Mostly to be symmetric, we also handle the right angle bracket, the "greater than," >, similarly. Single and double quotation marks are used to delimit attributes, so are part of the XML specification, but do not present difficulties in narrative text.

In normal writing, always use the empty elements <ampersand />, <less />, and <greater />. Inside of mathematics elements, or code for images written in LaTeX, always use the pre-defined macros, \amp, \lt, \gt. In verbatim text (such as programs) always use the XML entities &, <, >.

If you consistently follow the rules in the previous paragraph you will avoid a descent into escape-character hell and avoid a lot of head-scratching. In particular, you should have no need of the <![CDATA[ ]]> mechanism of XML, so just forget we even mentioned it.

Print and PDF output is generated via LaTeX, which has a good many special characters. So to preserve conversion to this format, you should consistently use provided empty elements for these characters. Here are the characters and their corresponding elements.

#	<hash />
$	<dollar />
%	<percent />
^	<circumflex />
&	<ampersand />
_	<underscore />
{	<lbrace />
}	<rbrace />
~	<tilde />
\	<backslash />

Table 3.12.1: LaTeX's reserved characters and their elements

There are some other empty elements, which are conveniences for certain characters, or sequences of characters, that are difficult or unusual in LaTeX and also somewhat obscure as Unicode characters. Two examples are the copyright symbol, ©, and constructions like the abbreviation "e.g." for *exempli gratia*. We will document these later.

3.13 Verbatim and Literal Text

Typesetting literal text, usually in a monospace font, can sometimes be tricky. For short bits of such text, as part of a sentence in a paragraph, use the <c> tag, which is short for "code." For much longer blocks of literal text, with line breaks that are to be preserved, use the <pre> tag, which is short for "pre-formatted".

For the content of a <pre> element, the indentation will be preserved, though an equal amount of leading whitespace will be stripped from every line, so as to keep the code shifted left as far as possible.

The behavior of these two tags is to preserve characters exactly. Certainly the ASCII character set will behave as expected, and Unicode characters will migrate successfully to output formats based on HTML. As mentioned in Section 3.12 the ampersand and left angle bracket will confuse the initial XML processing. So use the XML entities &, <, > to represent these characters to the XML processor, xsltproc. Now your verbatim text might possibly contain characters that will confuse LaTeX when converted to PDF for

print output. These characters are "\", "{", and "}", which are used to begin a macro, begin a group, and end a group (respectively). (Need to document how these are handled.)

3.14 Sage

Sage is an open source library of computational routines for symbolic, exact and numerical mathematics. It is designed to be a "viable free open source alternative to Magma, Maple, Mathematica, and Matlab." MathBook XML contains extensive support for including example Sage into your document.

A typical use of the `<sage>` tag is to include an `<input>` element, followed by an `<output>` element. The content of the `<input>` element may be presented statically in PDF output, or dynamically as a Sage Cell in an output format based on HTML. Of course, for output as a SageMathCloud worksheet, the Sage code is presented in the worksheet's native format.

The content of the `<output>` element is included in PDF output, but not in dynamic instances, since it can be re-computed. Notably, there is a conversion which pairs input and output into a single file in the format used by Sage's `doctest` framework. So if expected output is provided, it becomes automatic to identify when Sage has diverged from your expectations, and you can adjust your examples accordingly.

The Sage Cell Server can also be configured to interpret different languages, because Sage by default contains everything needed to evaluate code in these languages. This is done by providing a `@language` attribute, where possible values are `sage`, `gap`, `gp`, `html`, `maxima`, `octave`, `python`, `r`, and `singular`. The default is `sage`.

Note that the dynamic formats (including the Sage Cell) may run Sage "interacts," so that is possible to embed interactive demonstrations into your dynamic output formats.

3.15 Side-by-Side Panels

A `<sidebyside>` is a useful organization of elements in a horizontal layout, and so begins to blur the line between content and presentation. While we default to organizing information in a vertical sequence, it is often desirable to organize smaller elements adjacent to each other horizontally. Specifically, images, tabular, figures, tables, paragraphs may all be combined and there is some control over vertical and horizontal alignment. Captioning, both overall and individually, is especially flexible. See Section 6.12 for details.

3.16 Mathematical Results

Definitions, theorems, corollaries, etc. are supported by the tags: `<theorem>`, `<corollary>`, `<lemma>`, `<algorithm>`, `<proposition>`, `<claim>`, `<fact>`, `<definition>`, `<conjecture>`, `<axiom>`, and `<principle>`. Each may have a `<title>` (strongly encouraged), and then contains a `<statement>` which is a sequence of paragraphs and other elements. As appropriate, some of these elements (such as a `<lemma>`) may contain an optional `<proof>` (or several), while other elements may not have a `<proof>` (such as a `<conjecture>`).

A `<definition>` is a natural place to define notation as well (see Section 3.19), and to use the `<term>` tag to identify the terminology being defined.

In order to assist readers locating numbered items, these items are all numbered consecutively in a group that includes `<example>`s, `<remark>`s and inline `<exercise>`s.

3.17 Front Matter

In the beginning of your `<book>` or `<article>` you can have a `<frontmatter>` element that contains various items that would precede your first `<chapter>` or `<section>` (respectively). Possibilities include `<titlepage>`, `<colophon>`, `<biography>`, `<abstract>`, `<dedication>`, `<acknowledgement>`, `<foreword>`, and `<preface>`. Some of these may be duplicated (e.g. several prefaces for multiple editions), many of these items are restricted to books (e.g. a foreword), and some items are restricted to articles (e.g. an abstract). The DTD (⟨⟨dtd-info⟩⟩) will help you place them in the right order in your source. See Section 6.13 for details.

3.18 Back Matter

Similar to front matter, there is material you might wish to include after your book's final <chapter> or your article's final <section>. Possibilities to place in a <backmatter> include <appendix>, <references>, <index-part>, and <colophon>. There are empty tags you can place into an appendix to generate lists of notation, or lists of particular elements of your choice, such as a list of figures. A similar empty element actually generates the index, <index-list />. See Section 6.13 for details on the back matter generally, and Section 6.14 for more on automatic lists.

3.19 Index and Notation Entries

Construction of an index and a list of notation is accomplished by placing information into your text in the appropriate places in the right way.

The tags for an index entry will change soon, so we defer describing this precisely. These should be placed within the element that they describe. By this we mean that an <index> element can be placed within a <theorem> to refer to just that theorem, or it might be placed within a <subsection> to refer to that subsection. In this way, electronic versions of your work can have an index that is more informative than a traditional index that uses just page numbers.

A similar device is used to create a list of notation for a technical (mathematical) work. Place a <notation> element as close as possible to the place where notation is first introduced. If you use the <definition> tag for your definitions, then this is a very natural place to also introduce notation. Inside of <notation> use the <usage> tag to include a short example of the notation in use. This will be treated as mathematics, so imagine that it will be wrapped in an <m> tag and use LaTeX syntax. The <description> tag should contain a very short description in words of what the notation is for. So "center of a group" would be a good description to accompany the usage "$Z(G)$."

3.20 WeBWorK Exercises

It is possible to author WeBWorK automated homework problems directly within your source. A static version will be rendered in your LaTeX/PDF/print output, and a "live" version will be rendered into HTML output (though a student is not able to authenticate against a course). However, you can also extract *all* of the WeBWorK questions from a textbook into a single archive suitable for uploading into a traditional WeBWorK server. This is a big topic, so see the dedicated Chapter 9 for details.

3.21 URLs and External References

The <url> tag always requires an @href attribute. Usually this will be some external web page, or other resource. If the <url> is empty, then the value of the @href attribute will be the link text, typically in a monospace font. You can also provide your own content, using elements much like any other piece of text that would occur in a paragraph. Of course, there is no need to anticipate any problems with LaTeX output and special characters like a tilde. Use the character itself in the @href, and use the <tilde> element in the content (or wrap the content in the <c> element and use the literal character).

This element may also be used to link to external data files. See Section 6.15 for details.

3.22 Scientific Units

If you are writing about science or engineering, or even if you are not, there is extensive support for scientific units. So, for example, you could author a force in metric units as

```
<quantity>
    <mag>20.7</mag>
    <unit prefix="kilo" base="gram" />
```

```
    <unit base="meter" />
    <per  base="second" exp="2" />
</quantity>
```

This would be rendered as 20.7 $\frac{\mathrm{kg\,m}}{\mathrm{s}^2}$. More in Section 6.16.

Chapter 4

Processing, Tools and Workflow

This chapter explains in full detail how to combine your source file with an XSL stylesheet to produce output. It expands on the simple example in Chapter 2 and should also be read in conjunction with the chapter on the `mbx` script (Chapter 8).

4.1 Basic Processing

The executable program `xsltproc` implements Version 1.0 of the **eXtensible Stylesheet Language (XSL)**. This is a declarative language that walks the hierarchical tree of an XML source file, and for each element describes some output to produce before, and after, recursively processing the contained elements. (That is a simplified description.)

`xsltproc` is typically installed by default on Linux systems and as part of Mac OS. See the MathBook XML website for details for Windows systems. The most basic operation is to provide `xsltproc` with an XSL stylesheet from the MathBook XML distribution and an XML document of your creation that is valid MathBook XML. This is done at the command-line, inside of a terminal or shell. Describing command-line operations, along with file and directory management, is beyond the scope of this guide, so consult another resource if this is unfamiliar. So here is a hypothetical simple example:

```
rob@lava:~/mathbook$ xsltproc xsl/mathbook-html.xsl ~/books/aota/animals.xml
```

By default, `xsltproc` writes output to `stdout` (the screen), which you could redirect to a file, or you could use the `-o` switch to send the output to a named file. However, MathBook XML automatically writes to a file whose name is derived from the `@xml:id` attribute of the top-level `<book>` or `<article>` tag. If no such attribute is given the filename will be derived from `book-1` or `article-1`. All output is produced in whatever the current default directory is, so you will likely want to set this beforehand.

The `xsl` subdirectory of the MathBook XML distribution contains a variety of XSL stylesheets, which I will also refer to as **converters** or **conversions**. The ones that you will use as an author all have filenames of the form `xsl/mathbook-XXX.xsl`, where `XXX` is some indication of the output produced. Conversions to LaTeX or HTML output are the two most mature converters.

Note that authors are not responsible for creating XSL stylesheets. Stock conversions are part of the MathBook XML distribution, and anybody is welcome to assume a source document is valid MathBook XML and create new conversions to process it to existing, or as yet unimagined, formats.

4.2 Modular Source Files

For a large project, such as a book, you will likely want to split up your source into logical units, such as chapters and sections. `xsltproc` supports an include mechanism that makes this possible. Let us suppose that a section of your book on animals has a chapter on mammals with a section on monkeys. Then you need to do the following:

1. For the file containing the <chapter> tag for the chapter on mammals, place the attribute xmlns:xi="http://www.w3.org/ on the outermost tag in the file.

2. Within the <chapter> element for the chapter on mammals, add the line <xi:include href="monkeys.xml" /> to "pull in" the section on monkeys at that location. The @href attribute can point to a file in a subdirectory, but will be interpreted relative to the location of the file containing the mammal chapter element.

3. Add the switch --xinclude to your invocation of xsltproc.

Note that when you invoke xsltproc the default directory can be far away from your source, and the processor will locate all the component files of your project through the relative file locations in the @href attribute. Several comments are in order.

- Begin small and start a project *without* using modular files. Modularizing seems to add a layer of complexity that sometimes obscures other beginner's errors. So get comfortable with a single source file before branching out.

- I am forever forgetting the --xinclude switch. Empty output, or cryptic error messages, are your first clue to this simple, but common, mistake.

- The XML specification requires that a source file only contain a single outer-most element. So for example, two <chapter> elements cannot go into the same file as simultaneous outer-most elements.

- In practice, there is not a lot to be gained by creating a subdirectory structure mirroring your modularization—all your source files can go into one big directory and the XML hierarchy will take care of the organization. I do sometimes like to name my files accordingly, so for example chapter-mammals.xml and section-monkeys.xml.

The sample book in examples/sample-book amply demonstrates different ways to modularize parts of a project (but should not be taken as best practice in this regard). This guide, in doc/author-guide is a simple example of modular source files, and might be a good template to follow for your book.

4.3 Verifying your Source

A **Document Type Definition**, usually known as a **DTD** is a formal specification of an XML vocabulary (the allowed tags and attributes), and how they relate to each other. So, for example, the restrictions that say you cannot nest a <book> inside of a <chapter>, nor can you nest a <subsection> in a <chapter> without an intervening <section>, are expressed and enforced by the DTD. One of the beauties of the DTD is that it is written using a specific syntax and tools exist that use a DTD as input. In particular, a source MathBook XML file that conforms to the MathBook XML DTD is said to be **valid**. You should strive to always have valid source files and so you want to regularly verify that this is the case.

You can find the MathBook XML DTD in

```
schema/dtd/mathbook.dtd
```

You can read this file, and may eventually become adept at translating the syntax. (We plan to also create something very similar in the RELAX NG syntax, which is more expressive.) Better is to use the xmllint program to validate a source XML file against a DTD. If you have xsltproc on your system, then almost certainly you have xmllint, since they use the same underlying core library, libxml2. Here is the command you want to use to validate a source file. (The --xinclude switch is optional if you do not have modular files, Section 4.2.)

```
$ xmllint --xinclude --postvalid --noout --dtdvalid
/path/to/schema/dtd/mathbook.dtd ~/books/aota/animals.xml 2> dtd-errors.txt
```

You will need to use a pager (`more`, or `less`), or your editor, to review the results in `dtd-errors.txt` (or whatever you choose to call that file). Your goal is to have this output file always turn up empty. But if not, you will eventually become skilled at interpreting the error output and fixing your source.

The other beauty of a DTD is that you can supply it to a text editor (Section 1.2) and then you will get context-sensitive help that greatly assists you in using only the tags and attributes that are allowed in a given location of your source. XML Copy Editor is the one editor like this we have tried, but we do not have extensive experience.

Two caveats for your use of `xmllint`. First, the filenames in the error messages are not always accurate, especially if you are using the `xinclude` mechanism and your files are organized in subdirectories. In this case, or in the case of modular files and many, many errors, you can apply `xmllint` to just one file at a time and begin cleaning up errors that way. Second, a DTD is not quite as expressive as we would like, and so it is not a perfect description of the ideal MathBook XML source file. But if the DTD says something is *not* valid, than it almost certainly is.

You should regularly check that your source is valid (script the command above), and whenever `xsltproc` fails, or your output is not what you were expecting, check against the DTD first.

4.4 Customizations, String Parameters

There are some aspects of your output that are entirely divorced from the actual content, and are presumably all about how that content is presented. Two good examples are the size of the font used in LaTeX/PDF/print output, and the granularity of web pages in HTML output (by this we mean, is each web page a whole chapter, a whole section, a whole subsection?). Producing output with varying values of these parameters does absolutely nothing to change your content in any way, and so should not be a part of your source. Thus we provide values for these parameters on the command-line *at processing time*. They have become known as **stringparam** for a soon-to-be obvious reason.

Suppose you want to make a large-font version of your textbook for a student who has limited vision. Look inside the top of `xsl/mathbook-latex.xsl` and find the `latex.font.size` parameter. The preceding comments in this file suggest `20pt` is the maximum supported. So you would use a command-line like the following (possibly with `--xinclude`, etc.).

```
$ xsltproc --stringparam latex.font.size "20pt"
/path/to/xsl/mathbook-latex.xsl ~/books/aota/animals.xml
```

You can use as many stringparam as you like on the command-line (or in your scripts). The quotation marks are not strictly needed in this example, but if the value of the parameter has spaces, slashes, etc., then you need to quote-protect the string from the command-line processor, and either single or double quotes will work (and protect the other kind).

These parameters are documented in the XSL files themselves, principally `-common`, `-latex` and `-html`, and occur near the top. They assume sensible defaults for beginners, and error-checking is careful and robust. They will be easier to locate and use when we have the time to document them more carefully here in the Author's Guide.

One caveat for using these is that experience has taught us that some of the parameters we created early on really do affect your content. A good example is the `autoname` switch, since what you type as source is greatly influenced by what assumptions you have for the value of this switch, and you are unlikely to get decent output if you change it. So this decision should really be reflected in, and run with, the source. We will change some of these, but always provide a smooth upgrade path through deprecations, with little or no disruption to your workflow.

4.5 Customizations, Thin XSL Stylesheets

Stringparams (Section 4.4) are an easy way to effect global changes in the presentation of your writing. But putting ten of them on every command-line gets old and cumbersome fast.

You may also wish to customize your output in some stylistic way. This might be especially true for LaTeX/PDF/print output. For example, you might wish to have every chapter heading of your book in a nice

shade of light blue, with the title flush right to the margin, countered by a thick solid rule extending all the way right, to the edge of the paper. Notice that this does not affect your content, it is strictly presentation.

We have done several things to encourage such customizations. We have tried to put as much stylistic information as possible in the LaTeX preamble and keep as much as possible out of the body. (There is always room for improvement on this score, please be in touch if you have a need.) For small adjustments the `latex.preamble.early` and `latex.preamble.late` stringparam are possible vehicles, though all the LaTeX code to make light blue, flush-right rules is going to be messy on the command-line.

Instead, you can make a small XSL file, to use as input to `xsltproc`. The first thing it should do is import the stock MathBook XML file for the type of output you want to create. You can use an absolute path to the MathBook XML distribution (which will not be very portable), or utilize the `mathbook/user` directory and a relative path from there. The easiest thing to put in this file is elements like

```
<xsl:param name="latex.font.size" select="'20pt'" />
```

which is functionally equivalent to our example in Section 4.4. Values given on the command-line supersede those given in an XSL file this way.

You can augment the LaTeX preamble with as much LaTeX code as you like in the following way.

```
<xsl:param name="latex.preamble.late">
    <xsl:text>% Proof environment with heading in small caps&#xa;</xsl:text>
    <xsl:text>\expandafter\let\expandafter\oldp\csname\string\proof\endcsname&#xa;</xsl:text>
    <xsl:text>\let\oldep\endproof&#xa;</xsl:text>
   <xsl:text>\renewenvironment{proof}[1][\proofname]{\oldp[\scshape #1]}{\oldep}&#xa;</xsl:text>
</xsl:param>
```

There are a variety of things you can do generally, by overriding the imported XSL templates to change behavior, but such modifications are beyond the scope of this guide.

4.6 Images and the `mbx` Script

We believe it is important to preserve a record of how diagrams and other graphics are produced. This can be easy when a graphics language is employed to describe the graphical elements, rather than creating a bit-mapped image with some other interface. So we have `<asymptote>`, `<latex-image-code>`, and `<sageplot>` for elements holding code to produce diagrams or images.

The upside to this is that small edits to the code can easily accomplish minor changes or corrections necessary for the images. The LaTeX macros provided by an author can be used in the text *and* in a diagram, leading to greater consistency between the two. Finally, starting from source, we can do the best possible job of producing image formats that are compatible with the document output formats and which scale smoothly in PDFs and in web browsers.

The downside to this is that XSL is not a general purpose programming language, and so in particular, cannot call "helper" programs such as `asy`, `pdflatex`, and `sage`. The general strategy is to use XSL to identify and isolate the parts of a document that lie in the elements designed for graphics languages. A Python script, the `mbx` script, employs these XSL stylesheets and then feeds each image file to the appropriate helper program.

This script has a variety of options, so we document it fully in Chapter 8.

4.7 File Management

MathBook XML, at its core, is the formal specification of the XML vocabulary, as expressed in the DTD (Section 4.3). We have provided converters to process source files into useful output. However, we have not yet built a point-and-click application for the production of a book. So you need to take some responsibility in a large project for managing your files, both input and output. We have tried to provide flexible tools to make an author's job easier. The following is advice and practices we have successfully employed in several book projects.

Source I am fond of describing my own books with an initialism formed from the title. So *A First Course in Linear Algebra* becomes FCLA, and in file and directory names becomes `fcla`. So I have a top-level directory `books` and then `books/fcla`, but this directory is not the book itself, this is all the extra stuff that goes along with writing a book, much of it in `books/fcla/local`. The actual book, the part everybody sees with an open license, lives in `books/fcla/fcla`. This subdirectory has files like `COPYING`, which is a free software standard for license information, and `README.md` which is a file in the simplistic Markdown format that is picked up automatically by GitHub and displayed nicely at the book's repository's main page. Subdirectories include `src` for the actual XML files, `xsl` for any customizing XSL (Section 4.5), and `script` for shell scripts used to process the book (see below).

I do not use any additional directory structure below `src` to manage modular files for a book, since the XML and the `--xinclude` mechanism manage that just fine. I see little benefit to extra subdirectories for organization and some resulting inconvenience. I do typically have a single subdirectory `src/images` for raster images and other graphics files.

I believe it is critically important to put your project under revision control, and if licensed openly, in a public GitHub repository. So the `books/fcla/fcla` directory and all of its contents and subdirectories is tracked as a `git` repository and hosted on GitHub. Because this directory is *source* I try very hard to *never* have any temporary files in these directories since I do not want to accidentally incorporate them into the `git` repository. As a general rule-of-thumb, only original material goes in this directory and anything that can be re-created belongs outside.

A tutorial on `git` would be way outside the scope of this guide, but Beezer and Farmer *have* written *Git For Authors*, so perhaps look for that.

Image Files Some images are raster images (e.g. photographs) that are not easily changed, and perhaps unlikely to be changed. Other images will come from source-level languages via the `mbx` script. For your convenience, this script has a command-line option that allows you to direct output (graphics files) to a directory of your choice.

In the early stages of writing a book, I put image files produced from source code in a directory outside of what is tracked by `git`. It is only when a project is very mature that I begin to include completed graphics files into the `src/images` directory for tracking by `git`.

Build Scripts When you have a mature book project, the various files, processing options, and a desire for multiple outputs can all get a bit confusing. Writing simple scripts is a good idea and the investment of time doing this early in a project will pay off through the course of further writing and editing. The particular setup you employ is less important.

I have fallen into the habit of using the `make` program. It allows me to define common variables upfront (such as paths to the MathBook XML distribution and the main directory for the project it applies to). Then I can easily make "targets" for different outputs. So, for example I typically go `make pdf` or `make html` to produce output, and have simple companion targets so that I can go `make viewpdf` or `make viewhtml`. Other targets do things like checking my source against the DTD (Section 4.3). I have split out the variable definitions in a way that a collaborator can join the project and simply edit the file of definitions just once to reflect their setup, and still participate in future upgrades to the script by pulling from GitHub and not overwrite their local information.

My use of `make` is a bit of an abuse, since it is really designed for large software projects, with the aim of reducing duplicative compilations and that is not at all the purpose. You could likely have exactly the same effect with a shell script and a case (or switch) statement.

My general strategy is to assemble all the necessary files into a temporary directory (under `/tmp` in Linux) by copying them out of their permanent home, copy customizing XSL into the right place (typically `mathbook/user`), run the `mbx` script as necessary and direct the results to the right place, and finally copy results out of the temporary directory if they are meant to be permanent. Interesting, an exception to staging all these files is the source of the book itself which is only read for each conversion and then not needed for the output. So you can just point directly to a master file and the `xinclude` mechanism locates any other necessary source files.

A good example of this general strategy is the use and placement of image files for HTML output. It is your responsibility to place images into the location your resulting HTML files expect to locate them.

By default, this is a subdirectory of the directory holding the HTML files, named `images`. You will want to copy images, such as photographs, out of your main source directory (`src/images`?). But you may be actively modifying source code for diagrams, and you want to re-run the `mbx` script for each run, and make sure the output of the script is directed to the correct subdirectory for the HTML output. Running the `mbx` script frequently can get tiresome, so maybe you have a makefile target `make diagrams` that updates a permanent directory, outside of your tracked files in the repository, and you copy those files into the correct subdirectory for the output. That way, you can update images only when you are actively editing them, or when you are producing a draft that you want to be as up-to-date as possible. As a project matures, you can add images into the directory tracked by `git` so they are available to others without getting involved with the `mbx` script.

We did not say it would be easy, but we feel much of this sort of project management is outside the scope of the MathBook XML project itself, while in its initial stages, and existing tools to manage the complexity are available and documented. (We *have* been encouraged to create sample scripts, which we may do.) Just remember the strategy: stage necessary components in a temporary directory, build output in that directory, copy out desired semi-permanent results, and limit additions to the source directory to that which is original, or mature and time-consuming to reproduce.

4.8 Doctesting Sage Code

Adding computer code to your textbook is a tricky proposition. You can propose that it is merely an illustration, and not meant to have all the necessary details, or you can make it exact, correct and executable, and then risk inevitable changes to render your code obsolete. At least you have the option of editing and reposting online versions quickly and easily.

One of our main motivations for this project was mixing in code from the powerful, open source, mathematical software project, Sage (Section 3.14). When you add example Sage code to illustrate mathematical ideas, you are then encouraged to also include expected output in the `<output>` element. Here comes one of the powerful advantages of XML source and XSL processing.

The `mathbook/xsl/mathbook-sage-doctest.xsl` stylesheet, used in the usual way, will create one (or several, depending on the `chunk.level` stringparam) file(s), in *exactly* the format Sage expects for automated testing. So all your words are gone, and all your Sage input and output is packaged so Sage can run all the `<input>` and compare the results to the expected `<output>`.

We have many years' experience testing hundreds of non-trivial Sage examples from textbooks, for linear algebra and abstract algebra. Roughly every six months, we discover ten to twenty examples that fail. Frequently the failures are trivial (usually output gets re-ordered), but some are significant changes in behavior that leads us to re-word surrounding guidance in the text, and in a few cases the failures have exposed bugs introduced into Sage. It has been relatively easy to do this maintenance on a regular basis, and if it had not been done, the accumulated errors would be enough to greatly degrade confidence in the accuracy of the examples.

Exact details for this process can be found in Section 6.18. Note that Sage is really just a huge Python library, so it might be possible to test pure Python code with this facility, but we have not tested this at all. Similar support for other languages can be considered if requested for use in serious project.

4.9 Author Tools

The general stringparam `author-tools` may be set to `yes` to activate special handling of the `<todo>` element, the `@provisional` attribute of an `<xref>` element, and notation and index entries. The LaTeX-specific stringparam `latex.draft` set to `yes` will automatically activate the previous features, in addition to a few others appropriate to the printed page. We have an XSL stylesheet that dumps every `<todo>` and every `@provisional` attribute to the screen as a simply-formatted text report, but it is not public now.

The intent here is to make a rough draft, for an author or collaborator only, reporting as much as possible that is incomplete, pending or hidden in the usual output. But none of this is carefully supported. Requests for improvements, and overall comments, from working authors are invited. They will help to make this feature more useful for everybody.

Chapter 5

MathBook XML Syntax Specification

Mitchel T. KellerDepartment of Mathematics Department of Mathematics Washington and Lee University kellermt@wlu.edu

This MathBook XML Author's Guide, along with the sample article and sample book distributed with the MathBook XML source, provide a wealth of examples of how to author in MathBook XML. However, at some point, you will undoubtedly encounter a situation where some of your text fails to appear in your output or `xsltproc` produces an error. Those are good moments to start investigating the formal specifications of MathBook XML syntax, as most likely you tried to use something in a way incompatible with those specifications. This chapter will help you with the basics of reading these specifications.

5.1 Document Type Definition (DTD)

A **document type definition** (usually abbreviated as DTD) is a way of specifying the formal structure of an XML document, such as a MathBook XML file. It describes things such as the tags that are defined, where they may be placed relative to one another, and attributes that may be applied. DTDs were developed early in the evolution of XML, and thus they are widely supported. However, they have some limitations. In particular, there are some subtelties to MathBook XML that a DTD cannot capture, so RAB intends to eventually produce a Relax NG schema (see Section 5.2) that will cover these cases. Until then, we have the DTD.

5.1.1 Finding the DTD

The MathBook XML distribution contains the DTD in the `schema/dtd` directory. However, as a plain text file, it is rather hard to read. Thus, we suggest that you take a look at the version on the MathBook XML website that is displayed using LiveDTD, since it provides some nice clickable features. The version you get with the MathBook XML distribution is useful, however, as you can run the program `xmllint` to check your MathBook XML file for validity. See Section 4.3 for more information.

5.1.2 Reading the DTD

Let us start by taking a look at how the DTD specifies the definition of a section, which is included below, but if you wouldd like a more interactive version of it, go to the LiveDTD version of the DTD, scroll down to `<section>` in the left panel, and click on it.

```
<!ELEMENT section (title, index*, author*, (((%block;|references|exercises)*)|
    (introduction?,
        subsection,(subsection|references|exercises)*,conclusion?))) >
    <!ATTLIST section xml:id   ID       #IMPLIED>
    <!ATTLIST section xml:base CDATA    #IMPLIED>
    <!ATTLIST section xmlns:xi CDATA #FIXED "http://www.w3.org/2001/XInclude">
```

We will start with the first line. It starts off by telling us that we are defining an element called <section>, and everything from there to the end of the line tells us the composition of a <section>. First we see title, index*, author*,. This is telling us that the section *must* have a title, and <title> *must* come immediately after <section>. The * on <index> and <author> mean that the section title may be followed by zero or more <index> tags and zero or more <author> tags. (But do not think about putting <author> before <index>!) If you look at the source code for this chapter (which has a definition that begins identically to that for section), you will see that I (MTK) added an author tag to claim responsibility for any mistakes in this chapter for the time being. However, RAB did not put author tags in the other chapters, since he wrote them all, and so the overall author tag covers that.

The next section of the specification of <section> is where things get interesting. Carefully examining the parentheses, you will see that there are two groupings separated by |. This means that what comes after the <title>, <index>, and <author> must take one of two forms: (%block;|references|exercises)* or introduction?, subsection,(subsection|references|exercises)*,conclusion?. The * on the first option tells us that we may have zero or more occurrences of what comes inside the parentheses. Thus, we could go straight to a <section> tag, which might be a good idea when outlining a document before you start writing. However, assuming you want a positive number of the things in the parentheses, this part of the specification allows you to jumble up %block;s (whatever they are), <references> tags, and <exercises> tags to your heart's content. What is a %block;? If you click on the link for the LiveDTD, it will jump you to the definition of the block entity, and you will see that the things usuable here include theorem, corollary, fact, definition, exercise, etc. But there is also another mysterious thing beginning with a %: %para;. One quick click jumps us up and shows us that %para; is shorthand for <p>, the various lists, and a number of other things including <figure> and <table>. Thus, you can shove lots and lots of things inside a <section> (with a wide variety of orders permitted). However, there are some things *not* encapsulated by (%block;|references|exercises)*. For instance, it would not make sense to put a <chapter> inside a <section>, so that is not allowed. On the other hand, a <subsection> inside a <section> makes perfect sense. To see how we can do that, we need to look at the second part of the specification of <section>.

When we look at the second part of the specification of <section>, we encounter most of the rest of the key ideas of reading the DTD: introduction?, subsection,(subsection|references|exercises)*,conclusion?. The commas separating these parts indicate order is important, just as with the title, index*, author* part. The ? indicates that the <section> may have at most one <introduction>. There then *must* be a <subsection>, as indicated by the lack of decorations like ? or * on <subsection>. That mandatory <subsection> may then be followed by zero or more occurrences of <subsection>, <references>, and <exercises> in any order you wish. (As an example, each <subsection> could be followed by a block of <exercises> and a list of <references> before the beginning of the next <subsection>. Finally, we see another ? element: <conclusion>, which again is optional as indicated by the question mark.

After the !ELEMENT part of the definition of <section>, we see a list of !ATTLIST tags, which specify the attributes that can be assigned to <section>. The @xml:base and @xmlns:xi attributes are not particularly interesting, but @xml:id is useful. This tells us that we may assign an identifier to the <section> by writing something of the form <section xml:id="schema-dtd">, which then allows us to use the <xref> tag to insert a reference. (If you have a background in LaTeX, think of xml:id="blah" as being like \label{blah} and <xref ref="blah" /> as being like \ref{blah}, although with some additional features. See Section 3.3.

For another example, let us look at the specification of <exercise>:

```
<!ELEMENT exercise (title?, index*, ((statement, hint*, answer*,
    solution*)|(introduction?, webwork, conclusion?)))>
    <!ATTLIST exercise xml:id ID    #IMPLIED>
    <!ATTLIST exercise number CDATA #IMPLIED>
```

Here we see that an exercise may have at most one <title>, which can be followed by zero or more <index> tags. There are then two options: a <statement> (followed by zero or more <hint>s, followed by zero or more <answer>s, followed by zero or more <solution>s) or a singular <webwork> tag (optionally preceded by an <introduction> and/or followed by a <conclusion>). Again, the attributes include an @xml:id for referencing purposes. Here we also have the option of a @number attribute. See the Sample Article section on exercises for more information.

Let us look at one final example, which is the DTD entry for <ul>, the unordered list.

```
<!ELEMENT ul (li+)>
    <!ATTLIST ul cols  CDATA #IMPLIED>
    <!ATTLIST ul label CDATA #IMPLIED>
```

Here we see that there is exactly one thing that can be put inside a `<ul>` tag: `<li>`. The `+` after `<li>` tells us that we must have *at least* one `<li>` tag inside our list. That is, we may not have an empty list! The `@cols` attribute allows a list to be formatted in a way that it consists of multiple columns. The `@label` attribute determines what graphical markers are used for each list item. (Since these attributes are really metadata that describe how the MathBook XML file should be rendered into various output formats and not structural, the DTD does not give us this information, and we need to consult the documentation for MathBook XML to learn these facts.)

5.1.3 Where Can I Put That Tag?

After authoring in MathBook XML for a while, you will likely run into a situation where you put a lengthy passage in your source, wrapped in a particular tag, and the text never appears in your HTML or LaTeX output. Quite possibly, you have violated the DTD, and thus the call to `xsltproc` is silently dropping your text because it was not expecting it at that point. For example, suppose you tried putting an `<image>` inside a `<p>`. If you are lucky, you are using an editor such as emacs in nXML mode, which will highlight the `<image>` tag to tell you it is not valid. If you are unlucky, maybe `xsltproc` silently refuses to include your image in the output. (In between options include giving a warning or running `xmllint` to discover your MathBook XML is invalid. You may even find that, although your MathBook XML is invalid, `xsltproc` produces reasonable output. However, violating the MathBook XML specification means that you are not guaranteed to continue to get reasonable output in future updates, so you should correct the problem.) If you go to the LiveDTD and click on the + next to "image" in the left-hand column, you will get an entry telling you that the `<image>` element (tag) is seen in `<figure>` and `<sidebyside>` and the definition of the entity `%para;`. That tells you that if you want to have valid MathBook XML that will not lose your image, you need to put it inside a `<figure>` or `<sidebyside>` *or* anywhere that allows for a `%para;`. Ugh. What allows for the inclusion of the combination of things lumped together under `%para;`? At the top of the left frame of the LiveDTD, click on `% Entities`, and then click the `+` next to `para`. That will give you list of tags, including `<answer>`, `<proof>`, and `<statement>`, that can contain a `%para;` (and therefore an `<image>`). You will notice that `<p>` is nowhere to be found, which tips you off that you cannot nest an `<image>` inside a `<p>`, forcing you to edit your source.

Although not a comprehensive overview of the contents of the MathBook XML DTD, hopefully this brief tutorial has empowered you to dig into the DTD as needed. As a result, you should be well on your way to writing valid MathBook XML (or being able to ask useful clarifying questions on the Google Group).

5.2 (*) Relax NG

This will get written once we have a Relax NG schema for MathBook XML other than the one `trang` automatically generates for us from the DTD.

Chapter 6

(*) Topics

Careful descriptions arranged into topics. Unwritten sections provide a temporary outline.

6.1 (*) Exercises, Inline and Sectional

6.2 Verbatim and Literal Text

This section expands on parts of Section 3.13. For descriptions of more involved uses, such as program listings and console sessions, see Section 6.11.

The tags described here contain *only raw characters*. By that we typically mean the first 128 characters of the ASCII code. Unicode characters are likely to migrate to HTML output just fine, but results for LaTeX output will be variable. The restriction to character data has two consequences. First, any markup will be silently ignored. Second, you need to observe the rules on special characters and escaped characters for XML, which are mercifully simple.

> In your source, use `&` for an &, and use `<` and `>` for < and >. Otherwise, every other ASCII character will render faithfully across all possible formats.

6.2.1 Short, Inline, Verbatim Text

The `<c>` is a mnemonic for "code", but is really meant to be any chunk of literal characters that you want to emphasise that way. It is meant for use within a sentence or caption ("inline") so its use is limited to those situations, and others that are similar, such as a title or a cell of a table. Typically these pieces of text do not break across lines, so can bleed into right margins, or cause very short lines. So keep the content short, and/or use early in the first sentence of a paragraph where you know it will not affect a line break. For longer chunks, see options following.

Typical presentation is a monospace font, perhaps of a slightly heavier weight.

There is two caveats for use when creating LaTeX output. We have chosen to use the ? character as a delimiter for the start and end of a run of verbatim text. Unfortunately, such a choice is necessary. But if your run of characters includes a question mark, then the run will end prematurely. So we have an attribute, `@latexsep` that can be set to some other character, such as `!` or `|`, that does not appear in your content. This choice has no effect on other output formats.

The other caveat for LaTeX output is that you may use the `<c>` tag within a `<title>`, but you should avoid LaTeX's special characters (such as \, #, etc.). Regular characters should migrate to titles as verbatim text just fine, but we do not guarantee all possibilities.

6.2.2 Longer, Inline, Verbatim Text

For longer pieces of verbatim text, use the `<cd>` tag, which is short for "code display", analogous to the `<md>` for mathematics. It is used within sentences of a paragraph and will be presented with a vertical break above

and below, but without interrupting the paragraph. Because of the display presentation, it cannot be used other places, such as a <title>, where a vertical gap is not appropriate. All of the previous discussion about special characters applies for this tag.

You have two options in use. You may author inline with the rest of a sentence, with no extra newlines or whitespace before, after, or within the content. The result will be a single displayed line.

Or you may structure the content using one, or more, of the <cline> tag, which is meant to be similar to the <line> tag used elsewhere. You should still take care to not place any extra whitespace before or after the <cd> element, but inbetween the <cline> you may use as much visual formatting of your source as you wish, especially if you like your source to mirror your output. For LaTeX output there is less danger that your content will inadvertenly end the verbatim text prematurely.

6.2.3 Blocks of Verbatim Text

If you want to isolate large chunks of verbatim text outside of paragraphs, the <pre> tag is the one to use. It can be used as a peer of paragraphs (and other structures) as a child of a subdivision, or it can be placed into a <listing> to receive a caption, title and number.

You can structure the contents with <cline> in exactly the same manner as for <cd>. But you may find this tedious. Instead, you can make the content of <pre> a sequence of lines separated by newlines. So that you can preserve the indentation of your source, the line closest to the left margin is taken to actually be the left margin, and a corresponding amount of leading whitespace will be removed from every line. This will work well if you recognize two caveats. First, results will be unpredictable if you mix spaces and tabs for indentation. Sticking with spaces is best. Second, if your first characters of content immediately follow the <pre> tag then there is no leading whitespace and it is as if that line is already at the left margin. Then subsequent indentation may seem too severe to you.

As previously mentioned, Section 6.11 discusses the <console> and <program> tags which are more specific, and hence more capable. Review the possibilities before you decide between <pre>, <console>, and <program>.

6.3 (*) References and Citations

6.4 (*) Cross-Referencing Information

6.5 Subdivisions

A **subdivision** is a structured component of a book or article that would be recognized by most any reader. They are essential to the organization of a MathBook XML project. Notice that we use the generic term subdivision, since a <section> is just one example of a subdivision.

Subdivisions are <book>, <article>, <part>, <chapter>, <section>, <subsection>, <subsubsection>, and <paragraphs>. Their use is fairly intuitive, though there are some restrictions, so please read on.

A <book> must contain at least one <chapter>, which may contain <section>, <subsection>, and <subsubsection>. The tag <part> is planned for books, between <book> and <chapter>. It will function in two user-selectable ways: structural (e.g. numbering will reset), or decorative (merely inserting a decorative page between two chapters and sectioning the Table of Contents).

An <article> is simpler and shorter than a book. It might be really simple and have no subdivisions at all, or it may have <section>s. It cannot have <chapter>s, as that would be a <book>. Within a <section>, <subsection>s and <subsubsection>s may follow.

Subdivisions must nest properly and may not be skipped. So a <section> cannot contain a <chapter> and a <subsection> may not be contained in a <chapter> without an intervening <section>.

A subdivision *must* contain a <title>, and may contain one or more index entries (see ⟨⟨topic-index⟩⟩), which should appear before anything else. Any subdivision may be unstructured, with just a sequence of **top-level content** such as paragraphs, figures, lists, theorems, etc. Or a subdivision may be structured, and in this case it must follow a prescribed pattern. There may be a single, optional <introduction>, filled with top-level content, followed by a sequence of at least one of the appropriate subdivisions, ending with a single, optional <conclusion>, filled with top-level content. It is an error to begin with a run of top-level

content inside a subdivision and then begin to use subdivisions. (The solution is to make the initial content an <introduction> and/or one or several subdivisions.)

There are exceptions to the above. For one, <paragraphs> is an anomalous subdivision, as a sort of lightweight sectioning command. It may appear in any subdivision, at any location within a subdivision, it may not be subdivided further (it is a leaf of the document tree), it never gets a number, and its title is formatted in a subsidiary way. I especially like to use this in a two- or three-page <article> that has no other subdivisions at all. Typical presentation has the title in bold, without much change in font size (if at all), inline with the first paragraph, and perhaps a bit of vertical space as it begins and ends. Despite the name, it may contain more than just paragaphs, so may contain any top-level-content that would go in any other subdivision.

Two other anomalous subdivisions are <exercises> and <references>. These can function as a further subdivision of any other subdivision. So for example, an <exercises> could be a peer of several <section>s, contained within a structured <chapter>, and in this case would behave like the other <section>s with regard to numbering and presentation. Detail on allowed content and its behavior are in Section 6.1 and Section 6.3, respectively.

6.6 (*) Mathematics

This section is incomplete, newer features are being discussed as they are introduced.

Punctuation After Display Math If a chunk of displayed math concludes a sentence, then the sentence-ending punctuation should appear at the conclusion of the display. (And certainly not at the start of the first line after the display!) But do not author the punctuation within the mathematics element, put it afterwards, where it logically belongs.

More specifically, place a sentence-ending period (say) *immediately* after the closing of an <me>, <men>, <md>, or <mdn> element. MathBook XML will place the period in your output in the right place and in the right way. (By using LaTeX's \text{} macro, if you are curious to know the details.) Here is an example. The XML source

```
<md>
    <mrow>(a+b)^2</mrow>
</md>.  Now...
```

will render as

$$(a+b)^2.$$

Now...

Take notice of the requirement that the punctuation must be *immediately* after the closing tag of the math element, otherwise it will not migrate properly.

6.7 (*) Lists and their Labels

6.8 (*) Exercises and their Answers

6.9 Images

6.9.1 Raster Images

A **raster image** is an image described pixel-by-pixel, with different colors and intensities. Photographs are good examples. Common formats are Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG, JPG), which will both work with pdflatex and modern browsers. JPEG are a good choice for photographics since they are compressed on the assumption they will be viewed by a human, while PNG

is a lossless format and good for line art, diagrams and similar images (if you do not have vector graphics versions, see below).

To use these images, you simply provide the filename, with a relative path. A subdirectory such as `images` is a good choice for a place to put them. It is your responsibility to place these images where the LaTeX output will find them or where the HTML output will find them. The XML would look like:

```
<image source="images/crocodiles.png" width="50%" />
```

Typically you would wrap this in a `<figure>` that might have an `@xml:id` attribute for cross-references, with or without a caption. There is no `@height` attribute, so the aspect ratio of your image is your responsibility outside of MathBook XML. The `@width` attribute is a percentage of the available width of the text (outside of a `<sidebyside>` panel). Default width is typically 90%.

You may also provide a `<description>` which will aid accessibility for electronic formats. Keep such readers in mind and provide as much description as possible. Keep the markup simple, since this will typically migrate to an HTML attribute that cannot contain any structure. Be careful to avoid double-quotes. For example,

```
<image source="images/crocodiles.jpeg" width="50%">
    <description>Five crocodiles partially submerged in the shallows.</description>
</image>
```

6.9.2 Vector Graphics

An image is a **vector graphic** if the file describes the geometric shapes that constitute the image. So a simple diagram would be a good candidate, but a photograph would not. Popular formats are Portable Document Format (PDF) and Scalable Vector Graphics (SVG). You will get the best results with PDF images in LaTeX output and SVG images for HTML. The principal advantage of these formats is that they scale (big or small) smoothly, along with fonts. This is critical when you cannot predict the screen size for a reader of an electronic version.

Unless you describe these images with a language (see next subsection), you are responsible for providing the PDF and SVG versions. The `pdf2svg` utility is very useful if you have PDF images only. To use these images, you simply follow the instructions above, but do not include a file extension. This alerts the conversion to use the best possible choice. So presuming we had files `images/toad-life-cycle.pdf` and `images/toad-life-cycle.svg`, an example would be:

```
<image source="images/toad-life-cycle" width="85%">
    <description>Diagram of the four stages of a toad's life.</description>
</image>
```

6.9.3 (*) Images Described by Source Code

To be written once elements and tags solidfy, see sample article for examples.

6.9.4 Image Archives

As an instructor, you might want to recycle images from a text for a classroom presentation, a project handout, or an examination question. As an author, you can elect to make images files available through links in the HTML version, and it is easy and flexible to produce those links automatically.

First, it is your responsibility to manufacture the files. For making different formats, the `mbx` script can sometimes help (Chapter 8). The Image Magick `convert` command is a quick way to make raster images in different formats, while the `pdf2svg` executable is good for converting vector graphics PDFs into SVGs. Also, to make this easy to specify, different versions of the same image must have identical paths and names, other than the suffixes. Finally, the case and spelling of the suffix in your MBX source must match the filename (e.g. `jpg` versus `JPEG`). OK, those are the ground rules.

For links for a single image, add the `@archive` attribute to the `<image>` element, such as

```
<image  ...  archive="pdf svg">
```

to get two links for a single image.

To have every single image receive an identical collection of links, in `docinfo/images` place an `<archive>` element whose content is the space-separated list of suffixes/formats.

```
<archive>png JPEG tex ods</archive>
```

will provide four links on every image, including a link to an OpenDocument spreadsheet.

For a collection of images that is contained within some portion of your document, you can place an `@@xml:id` on the enclosing element and then in `docinfo/images` place

```
<archive from="the-xml-id-on-the-portion">svg png</archive>
```

to get two links on every image *only* in that portion (chapter, subsection, side-by-side, etc.). The `@from` attribute is meant to suggest the root of a subtree of your hierarchical document. If you use this, then *do not* use the global form that does not have `@from`.

You may accumulate several of the above semi-global semi-local forms in succession. An image will receive links according to the last `<archive>` whose `@from` subtree contains the image. So the strategy is to place general, large subtree, specifications early, and use refined, smaller subtree specifications later. For example,

```
<archive from="the-xml:id-on-a-chapter">svg png</archive>
<archive from="the-xml:id-on-the-introduction">jpeg</archive>
<archive from="the-xml:id-on-a-section-within" />
```

will put two links on every image of a chapter, but just one link on images in the introduction, and no links at all on every image image within one specific section. Again, do not mix with the global form. You can use the root document node (e.g. `<book>`) for `@from` to obtain a global treatment, but it is unnecessary (and inefficient) to provide empty content for the root node as first in the list—the same effect is the default behavior.

Notice that this facility does not restrict you to providing files of the same image, or even images at all. You could choose to make data files available for each data plot you provide, as spreadsheets, or text files, or whatever you have, or whatever you think your readers need.

Finally, "archive" may be a bit of a misnomer, since there is no historical aspect to any of this. Maybe "repository" would be more accurate. Though for a history textbook, it might be a perfect name.

6.10 (*) Tables and Tabulars

6.11 (*) Program Listings

6.12 (*) Side-by-Side Panels

6.13 (*) Front and Back Matter

6.14 (*) Automatic Lists

6.15 URLs and External References

6.15.1 URLs to External Web Pages

The `<url>` tag can be used to point to external items (as distinct from other portions of your current document, which is accomplished with the `<xref>` element, Section 3.3). It always needs a value for the `@href` attribute, most likely a URL. Most of this time, this will point to some resource available on the Internet but it could be a file on the system hosting your document, perhaps using a relative address (but see the rest of this section for some cautions). Here are three scenarios:

- The <url> element is empty. Then the value of the @href is also used for the visible text of the link, verbatim, and usually in a monospace font. Use **percent-encoding** (aka **URL encoding**) for the @href attribute to include special characters, such as spaces.
- The <url> element has content. Now the content should be authored as you would any other text in a sentence. Potentially problematic characters, such as a tilde should be authored with provided empty elements, but authored literally in the @href attribute.
- The <url> element has content, but you want this content to look like a URL. Use the <c> element around the content, and follow the rules for verbatim text. I do this often for simple URLs that point to the top level of a website. The @href is a complete URL like http://mathbook.pugetsound.edu/ but for content I use a less-imposing reader-friendly version like mathbook.pugetsound.edu.

For LaTeX output it gets quite tricky to handle all the various meanings of certain escape characters in URLs in more complicated contexts (such as tables, footnotes, and titles), so there may be some special cases where the formatting is off or you get an error when compiling your LaTeX. We handle most of these situations, but we always appreciate reports of missed cases.

6.15.2 URLs to External Data Files

The <url> element can be used to make data files available to your reader. Consider the example of a spreadsheet containing a large data set that a reader needs to analyze as part of an exercise. Here are our recommendations on how to accomplish this:

- If the file is hosted on some server unassociated with your project, and does not have a license compatible with your project, then just set the @href to the complete address. Be sure to include enough of the address for the reader of a a print version to be able to type in the URL, either as the content of the <url> or in close vicinity.
- If you authored the spreadsheet, or you are allowed to legally copy and distribute it, then place it on your server where you host your book project. Then do as above and use the full URL for the @href attribute, with a visible version available for PDF and print versions.
- If you have control over the placement of the file, you can host it on your server, and use a URL relative to the location of your HTML, PDF, or other files that comprise your document. This might be a good choice if your book will be posted many places and you can give it to others as an archive, like a *.zip file. It is a bad idea if a reader downloads a PDF without the data file following along and remaining in the same relative location. It is an impossible idea if your document gets printed on paper and there is no idea what a relative URL means and there is not even a link to click on.

Consider your audience and think about how much guidance they need about using context menus or helper/viewer applications to make use of the file formats you are providing. This advice may be different depending on the type of files and the types of output for your document.

6.16 (*) Units of Measure

6.17 Unicode Characters

MathBook XML supports (and encourages) the use of Unicode characters. Here are some relevant comments.

- Unicode characters will migrate well to any output format based on HTML. Most browsers will have a variety of fonts with glyphs to realize these characters.
- LaTeX will not always behave as smoothly. For openers, you definitely will want to use the xelatex engine to build a PDF. Then you need to be sure your system has a font with the necessary characters and you make the font known to xelatex. We are working out the details of the best way to accomplish this.

- How do you get a Unicode character into your source? In part this is specific to your operating system and editor, so is outside the scope of this guide, but we have hints below for popular operating systems.
- You can always place a Unicode character in your source using XML syntax. The first thing an XML parser will do is convert this syntax into a character. The number of the `SECTION SIGN` in hexadecimal is `A7`, so the syntax `&#xA7;` is identical to the character §. Of course, this will get tedious fast.
- The Full Unicode Input utility at `www.cs.tut.fi/~jkorpela/fui.html8` will allow you to specify a chunk of 256 consecutive Unicode numbers and then you can click on characters to make a string of several or many. You can cut/paste these into your source, or convert the whole lot to XML syntax all at once.
- Unicode characters have standardized names. You can find these, and more information, including font support, at the Unicode section of FileFormat.info, `www.fileformat.info/info/unicode/`. If you are struggling to find a specific character, then using this site's name in a search will often quickly locate what you need. Be sure to experiment with the test pages there for browser and font support (including checking your local configuration).
- ***Warning***: do not use Unicode characters as a way to get mathematical symbols (that is delegated to our use of LaTeX syntax). And do not use Unicode when we have provided an empty element for a character, especially when that character may be used in a markup syntax for some output, such as LaTeX, HTML, JSON, Markdown, ...

  For example, if you put many naked hash symbols (#) in your source, then you will get nice HTML, but when you try to get print from a PDF from LaTeX you will have a train wreck on your hands when you compile the LaTeX. Instead, be sure to always use the provided `<hash />` element. *Always*. Other empty elements are conveniences, which spare you from looking up Unicode numbers and make your source more readable, rather than a necessity to avoid special characters. An example is `<times />`, for use outside of a strictly mathematical setting: "I bought a 2×4 at the lumberyard."

6.17.1 Unicode Support in OSX

Mitch Keller reports on 2017-01-12 a way to get some popular characters with OSX. Use the Keyboard preference pane under System Preferences. In there, you can enable

```
Show Keyboard, Emoji, & Symbols Viewers in menu bar
```

Once you activate the keyboard viewer, you get a keyboard on your screen. When you hold down `opt`, it shows you what other symbol you would get if you push `opt+letter`. For instance, `opt+w` gives an upper-case Greek sigma and `opt+=` gives a not-equals sign (neither of which we can handle when processing the latex version of this guide). To get ä, you type `opt+u` and then hit `a`. This is illustrated by the keys for diacritical marks being highlighted in orange while holding opt. The shift key can have an effect to produce variations of some characters, such as quote marks (dumb versus smart).

6.17.2 (*) Unicode Support in Linux

6.17.3 (*) Unicode Support in Windows

6.18 (*) Testing Sage Examples

6.19 Building Output in SageMathCloud

SageMathCloud has *all* the tools you need to author with MathBook XML. You will need an upgrade from a subscription to allow Internet connectivity, but at a minimum a colleague with a paid plan can spare you one, they are plentiful and meant to be shared.

- Text editor: reasonably good, partially XML syntax-aware.
- `git`: installed (so clone MathBook XML).

- LaTeX: installed with many additional packages.
- Python: installed, necessary for `mbx` script.
- PDF viewer: handed off to your browser.
- HTML viewer: convert to the "raw" URL and you can preview.
- HTML server: nope. Zip up output and host elsewhere.

Chapter 7

(*) Add-Ons

Some features of the HTML version of a document rely on third-party services. Once an author configures them, then MathBook XML will do the rest for you. This chapter provides guidance on the configuration processes.

7.1 (*) Analytics

7.2 Search

Search facilities are enabled through Google Custom Search Engine. Please, please report any discrepancies in the following instructions as the setup interface at Google changes out from underneath us. These instructions are accurate as of 2016-12-12.

Besides being useful for search facilities, setting up a search engine might be a good way to alert Google of something newly available, and initiate your book's rise up the search results rankings.

List 7.2.1 (Configuring Google Custom Search)**.**

1. Create an account with Google (GMail, YouTube, etc.) and make sure you are signed in.
2. Visit GCSE and `Add` a new search engine of follow `New Search Engine`.
3. Provide a URL for the top-level domain name/directory for your book/document. Everything below this will be indexed. We have taken some care to mark knowl content in a way compatible with the search facility, but there is more work to do here.
4. Give the engine a GCSE-specific name, so you can tell later which one it is when you have several.
5. Under `Edit Search Engine > Setup > Basics > Details > Search engine ID` find a string which uniquely identifies your new search engine. Save this, you'll need to make it part of your MBX document.
6. Under `Edit Search Engine > Setup > Admin` add co-authors or trusted backup personnel.
7. Under `Edit Search Engine > Business > Settings` set your `Advertising status` to the non-profit setting if you qualify (most universties should).
8. Fiddle with `Edit Search Engine > Look and Feel` at your own risk! Only the defaults are tested and supported.
9. `Edit Search Engine > Setup > Indexing` sends you to Google Search Console to see if your book is already being indexed. YOu may need to go through a confirmation process to establish that you are the owner of the website being indexed. If you see taht your book is not yet being indexed, you may want to wait as long as a week before your material does get indexed and you make a search box available.

List 7.2.2 (Configuring MathBook XML for Google Search)**.**

1. The `Search engine ID` you saved from above is referenced in Google's code as a `cx` number. Add an element in your MBX source as `docinfo/search/google/cx` with the value of your book's ID as the content. See the sample article for a working example to mimic.

2. The `cx` element will alert the MathBook XML conversion and fully enable and implement search. You're done, and everything should just work. You should see a Google-branded search box to the top right of each of your pages. (We have no control over the branding.)

3. Time to rebuild your official HTML output and make the improved version available.

7.3 (*) Annotation

Chapter 8

The `mbx` Script

XSL is a very powerful language for text processing. However, it cannot do everything. The `mbx` script is a Swiss Army Knife of sorts to operate on parts of your document and manage processing that requires the application of external programs, such as LaTeX and Sage.

8.1 (*) Rough Quickstart

Some quick preliminary hints. This section *will* be expanded.

- `mbx -h`: help message, command summary
- `mbx -v`: progress indicators (verbose)
- `mbx -vv`: debugging information (doubly-verbose)
- Provide complete debugging output with bug reports

Much like the build advice at the end of Section 4.7, the `mbx` script collects necessary bits into a system-created temporary directory, does its work, and copies out the desired results. Some insight into failures can be found in this directory (which we leave behind for the system to clean-up later). Early in the `-vv` doubly-verbose output, this directory is reported after the string `temporary directory:`.

An example:

```
$ ~/mathbook/script/mbx -vv -c sageplot -f svg -d images ~/mathbook/examples/sample-article/sample-article.xml
```

This will extract every image described in `sample-article` by a `<sageplot>` element and produce output as SVG files (if possible), which will be deposited in the `images` subdirectory of the current working directory.

8.2 Restricting the Scope

The `-r` (`--restrict`) switch deserves special mention. It is followed by the value of an `@xml:id` attribute present in your source XML file. Then whatever action the script is asked to perform, it will only act on a subtree of the hierarchy, rooted at the element with the given `@xml:id` value.

So if your images are complex or numerous (or both!) and take a long time to process, you can restrict attention to whatever part of the document you are actively editing, and you can even restrict to a single `<image>` and so produce just a single graphics file.

8.3 `mbx` on Windows

At present, the `mbx` script assumes that your installation is similar to the one described in Appendix D. Making the `mbx` script Windows-compatible is an ongoing project. It is possible you may find a bug, which

we would ask that you report. In addition, if you have followed the directions in Appendix D, then you will need to customize the `mbx.cfg` configuration file, which tells `mbx` where to find the helper programs it relies on. See below for the details.

The script uses a LaTeX utility called `pdfcrop` to, well, crop PDF images generated by a LaTeX engine. This utility has not been made to work in the Windows CMD shell. If you want to generate images on Windows, you should use the Git Bash shell as described in Section D.3.

1. Copy the file

   ```
   /path/to/mathbook/script/mbx.cfg
   ```

   to

   ```
   /path/to/mathbook/user/mbx.cfg
   ```

2. Replace the right-hand side of the entry for `pdfpng` with the full path name of the `convert` utility installed with ImageMagick (Section D.5), ***using forward slashes***. It will be something like

   ```
   c:/ImageMagick-7.0.1-Q16/convert.exe
   ```

8.4 Python `requests` Library

In some situations the `mbx` script will go out on the Internet to fetch some interesting bits for you, saving you the trouble. These include

- Grabbing, downloading, and organizing stock thumbnails for YouTube videos, using a standard API provided for this purpose. These get used in PDF output in place of embedded videos.
- WeBWorK problems that are stored on a remote server will give up LaTeX versions if asked. Again, these are used for PDF output in place of interactive versions.

We use the Python `requests` module/library to manage the connections to these external servers. There are two items to be aware of.

Installation This library is not available on Apple computers by default. From Alex Jordan comes the following incantation at the command line,

```
sudo easy_install pip
sudo pip install requests
```

which was last tested 2017-03-31. Please update us if the situation has changed or there is more to add here.

Warnings Using this library to connect to a webserver securely via `HTTPS` will raise a warning since the support for SSL certificates is not complete. You will see messages similar to

```
site-packages/requests/packages/urllib3/connectionpool.py:843:
InsecureRequestWarning: Unverified HTTPS request is being made.
Adding certificate verification is strongly advised.
See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#ssl-warnings
InsecureRequestWarning
```

We have not figured out the best way to stop these, as of 2017-04-01.

Chapter 9

WeBWorK Automated Homework Problems

Alex Jordan

With a WeBWorK server (version 2.12 or higher, limited support with 2.11) and a little setup work, you can embed WeBWorK exercises in your MBX project. HTML output will have interactive problem cells. PDF output will contain static versions of exercises. And all such exercises can be archived by the `mbx` script into a file tree to be uploaded onto the WeBWorK server for use in the "traditional" way.

9.1 Configuring a WeBWorK Course for MBX

We assume a mild familiarity with administrating a WeBWorK server. The version of WeBWorK needs to be 2.12 or later for use with MBX, although with version 2.11 all features should function except for latex output with server-based problems (see Subsection 9.3.2. Using the `admin` course, create a course named `anonymous`. In the course's Course Configuration menu, set all permissions to `admin` (or perhaps set some to the even more restrictive `nobody`). Except set "Allowed to login to the course" to `login_proctor`.

In the Classlist Editor, add a user named `anonymous`, and set that user's permission level to `login_proctor`, the permission level one higher than `student`. Set that user's password to `anonymous`. Note that because this is public information, anyone will be able to log into this course as user `anonymous`. This is why setting the permissions earlier is very important. (Especially preventing this user from changing its own password.)

Add the following lines to the `course.conf` file (which lives in the parent folder of the `templates/` folder.)

```
# Hide message about previewing hints and solutions for instructors
$pg{specialPGEnvironmentVars}{ALWAYS_SHOW_HINT_PERMISSION_LEVEL} = 100;
$pg{specialPGEnvironmentVars}{ALWAYS_SHOW_SOLUTION_PERMISSION_LEVEL} = 100;
```

In the `templates/macros/` folder, edit `PGcourse.pl` (or create it if need be) and add the lines:

```
#### Replace essay boxes with a message
sub essay_box {
    my $out = MODES(
        TeX => '',
        Latex2HTML => '',
        HTML => qq!<P>If you were logged into a WeBWorK course
        and this problem were assigned to you,
        you would be able to submit an essay answer
        that would be graded later by a human being.</P>!
    );
    $out;
```

```
};

#### Suppress essay help link
sub essay_help {};

#### How many attempts until hint is available
$showHint = -1;
# May be a bug that WeBWorK requires -1 instead of 0
# for immediate access to hints

1;
```

Now both `xsltproc` with `mathbook-html.xsl` and the `mbx` script will be able to communicate with this course to retrieve what is needed.

9.2 WeBWorK Problems in Source

A `<webwork>` tag must be inside an `<exercise>`, optionally preceded by an `<introduction>`, and optionally followed by a `<conclusion>`.

```
<exercise>
    <introduction>
    </introduction>

    <webwork>
    </webwork>

     <conclusion>
     </conclusion>
</exercise>
```

There are several methods for putting content into the `<webwork>`. (Note that an empty `<webwork>` with no attributes will simply produce the camelcase WeBWorK logo.)

9.2.1 Using an Existing WeBWorK Problem

If a problem already exists and is accessible from the `anonymous` course's `templates/` folder, then you can simply include it as a `@source` attribute. For example, if it is a problem in the Open Problem Library (OPL) then relative to the `templates/` folder, its path is `Library/...` and you may use:

```
<webwork source="Library/PCC/BasicAlgebra/Exponents/exponentsMultiplication0.pg" />
```

Or if you have a problem's PG file, you can upload it into the `anonymous` course's `templates/local/` folder and use it with:

```
<webwork source="local/my_prolbem.pg" />
```

9.2.2 Perl-free Problems

If you'd just like to rattle off a quick question with no randomization, you can do as in this example:

```
<exercise>
    <webwork>
        <statement>
            <p><m>1+2=</m><var name="3" width="5" /></p>
        </statement>
    </webwork>
</exercise>
```

The above example could be given an optional title, introduction, conclusion, hint, and solution. These are discussed in Subsection 9.2.3.

In the above example, `"3"` is the `@name` attribute to a `<var>` element. This is how to create an answer blank that is expecting 3 as the answer. What you give as a `@name` attribute will be passed as a string to PG's `Compute()` command, so it needs to be valid input for `Compute()`. It should *not* begin with a `$`.

The default context is `Numeric`, which understands numerical expressions and formulaic expressions in the variable x. You can activate some other context as in this example:

```
<exercise>
    <webwork>
        <setup>
            <pg-code>
                Context("ImplicitPlane");
            </pg-code>
        </setup>
        <statement>
            <p>The answer is <m>x+y=1</m>.</p>
            <p><var name="x+y=1" width="8" /></p>
        </statement>
    </webwork>
</exercise>
```

Many special contexts are automatically detected by MBX, and it loads the appropriate macro file into the PG problem. However you may need to explicitly load a macro file as described in Subsection 9.2.3.

You should only use this (nearly) Perl-free shortcut if the `@name` attribute could be put in math mode and simultaneously serve as a printed answer in something like a solutions manual. In the above two examples, this is the case. But in a question with, say, `@name="cos(x)/2"`, if you'd like the solutions manual to print using `\frac{\cos(x)}{2}`, then you should write the problem as described in Subsection 9.2.3.

Additionally, in shortcutting around the structure described in Subsection 9.2.3, you will not have access to certain other features, such as answer format help links.

9.2.3 PG code in Problems

To have randomization in problems or otherwise take advantage of the algorithmic programming capabilities of Perl and WeBWorK's PG language requires using a `<setup>` tag. Having at least a little familiarity with coding problems in WeBWorK is necessary, although for simpler problems you could get away with mimicking the sample article in `mathbook/examples/webwork/`. A `<statement>`, (optional) `<hint>`, and (optional) `<solution>` follow. The whole thing can have an optional `<title>`.

```
<webwork>
    <title>Optional</title>

    <setup>
    </setup>

    <statement>
    </statement>

    <hint>
        <p>Optional</p>
    </hint>

    <solution>
        <p>Optional</p>
    </solution>
```

```
</webwork>
```

The <setup> contains a section of <var> tags followed by a <pg-code>. If you are familiar with code for WeBWorK PG problems, the <pg-code> contains lines of PG code that would appear in the "setup" portion of the problem. Typically, this is the code that follows TEXT(beginproblem()); and precedes the first BEGIN_TEXT or BEGIN_PGML. If your code needs any special WeBWorK macro libraries, you may load them in a <pg-macros> tag prior to <setup>, with each such .pl file's name inside a <macro-file> tag. However many of the most common macro libraries will be loaded automatically based on the content and attributes you use in the rest of your problem.

For each perl variable (scalar, array, or hash) that is used in the <pg-code> and which will *also* be used in the <statement>, <solution>, or as an answer to an answer blank, there should be a <var>. These <var> tags are primarily to help MBX handle static output, but they also allow for some optimal leveraging of WeBWorK features.

A <var> in the <setup> always has a @name attribute, which should match the variable's name in your <pg-code> (e.g. $x, @a, etc.). Each <var> should usually have a <static> tag with LaTeX code for the static version of the answer (possibly inside a \text{}). For PDF and other static output modes, this LaTeX code will be used to print <var> values, since the WeBWorK server will play no role.

Lastly, a <var> in the <setup> can have a @category attribute. This is intended for variables which will be used as answers. Based on a @category, an automatic help syntax link will be provided adjacent to an answer blank. For instance @category="point" will provide a link explaining the syntax for typing a point.

Here is a small example. Following the example, we'll continue discussing <statement> and <solution>.

```
<webwork>
    <title>Integer Addition</title>

    <setup>
        <var name="$a">
            <static>9</static>
        </var>
        <var name="$b">
            <static>8</static>
        </var>
        <var name="$c" category="integer">
            <static>17</static>
        </var>

        <pg-code>
            $a = Compute(random(1, 9, 1));
            $b = Compute(random(1, 9, 1));
            $c = $a + $b;
        </pg-code>
    </setup>

    <statement>
        <p>Compute <m><var name="$a" />+<var name="$b" /></m>.</p>
        <p>The sum is <var name="$c" width="2" />.</p>
    </statement>

    <solution>
        <p><m><var name="$a" />+<var name="$b" />=<var name="$c" /></m>.</p>
    </solution>
</webwork>
```

Within a <statement>, <hint>, or <solution>, reference <var> tags by @name. For HTML and PG output, the Perl variable will be used. For static output, the <var> tag's static child will be used.

Within the <statement>, a <var> tag with either a @width or @form attribute creates an input field. The @name attribute declares what the answer will be.

An <var> can have @form="essay", in which case it need not have a @name attribute. This is for open-ended questions that must be graded by a human. The form field will be an expandable input block if the question is served to an authenticated user within WeBWorK. But for the WeBWorK cells in MBX HTML output, there will just be a message explaining that there is no place to enter an answer.

An <var> can have @form="array". You would use this when the answer is a Matrix or Vector MathObject (a WeBWorK classification) to cause the input form to be an array of smaller fields instead of one big field.

An <var> can have @form="popup" or @form="buttons". These are not necessary for HTML and PG output to behave, but are needed if you intend for PDF output to emulate these answer entry field types.

If you are writing a multiple choice question and using @form="popup" or @form="buttons" in your <var>, instead of a <static> in the corresponding <var> from the <setup>, use a <set> tag, with <member> children. The <member> tags would be the multiple choice options, and each can have a @correct="yes" attribute to identify the correct choice(s). There is some unavoidable redundancy between listing these <member> tags in the <setup> and listing them again in the actual <pg-code>.

If you are familiar with PG, then in your <pg-code> you might write a custom evaluator (a combination of a custom answer checker, post filters, pre filters, etc.). If you store this similar to

```
$my_evaluator = $answer -> cmp(...);
```

then the <var> can have @evaluator="$my_evaluator".

9.2.4 Reusing a <webwork> by @xml:id

Planned.

9.3 Processing

9.3.1 xsltproc

If your project has <webwork> tags, then when you execute xsltproc, pass a webwork.server string parameter to it specifying where the server is that will do your processing. Example:

```
$ xsltproc --stringparam webwork.server https://webwork.myschool.edu <xsl> <xml>
```

If your webwork.server is running version 2.11 (the first version which can be used with MBX), then you should additionally pass --stringparam webwork.version 2.11.

For HTML output, this is all that is needed. For LaTeX, you may need to do more first. See Subsection 9.3.2.

9.3.2 LaTeX output

If your project uses PG files that live on the WeBWorK server, or if you have <webwork> tags that have image creation code in their <pg-code> tag, then you will need to retrieve LaTeX chunks from the server before you use xsltproc. The mbx script handles this, but only for WeBWorK 2.12 and later:

```
$ mbx -c webwork-tex -s https://webwork.myschool.edu -d <storage location> <xml>
```

The storage location would typically be a folder called webwork-tex/ located inside wherever you are having xsltproc put your output. Then when you run xsltproc, tell it where to access this content:

```
$ xsltproc --stringparam webwork.server.latex <storage location> <xsl> <xml>
```

9.3.3 Creating Files for Uploading to WeBWorK

All of the <webwork> that you have written into your project can be "harvested" and put into their own .pg files by the mbx script (this excludes <webwork> tags where you gave a @source attribute.) These files are created with a folder structure that follows the chunking scheme you specify. This process also creates set definition files (.def) for each chunk. For example, you might specify to chunk by section, and then you will have a .def file for each section, listing all of the .pg files associated with that section. For <webwork> tags that used a @source attribute, the .def file will include them as well. Lastly, this archiving process creates .pg files to be used as set header files to go along with each set definition.

```
$ xsltproc --stringparam chunk.level 2 <path to mathbook-webwork-archive.xsl> <xml>
```

This creates a folder called local/ that will have a subfolder corresponding to your project, which in turn has a folder tree with all of the .pg and .def files laid out according to your chunk level. You can tarball this local/ folder (compress it into a .tgz file and upload it into an active WeBWorK course where you may then assign the sets to your students (and modify, as you like).

Appendix A

FAQ: Frequently Asked Questions

This is a list of answers to frequent questions, in no particular order.

Why is there no tag for bold? That would be presentation, not content. I will answer that question with a question: *why* do you want to print something in bold? Is it emphasis? (See `<em>`.) Is it the volume number of a journal? (See `<journal>`.) Do you want to SHOUT? Try `<alert>`. And so on. There are lots of good answers, some of which are not yet implemented. We would love to hear about elements you need that are about expressing content, and not about altering presentation. See Item 1.1.1:1 in the MathBook XML Principles.

Why does your conversion to HTML use a fixed width for the text? There is an optimal number of characters per line for human readers, based on research and centuries of book design. So we set a fixed width such that that the default font comes close to achieving this optimal value. We also use responsive design to accomodate the constraints of a small screen as best as possible. A reader will not want to have to carefully resize a browser window to achieve the optimal width, nor should a line of text spread to many, many characters across a very expansive screen. See Item 1.1.1:4 in the MathBook XML Principles.

Everything looks right, but why do I get empty output and some warnings about bugs? There is a good chance you have "modularized" your source files and have not included the `--xinclude` switch on the command-line when you ran `xsltproc`. (See Section 4.2.) We can often recognize this mistake—did you not get a warning? (If not, we can improve the warning if you tell us how your source was organized. So please do, since we would love to hear about it.)

How do I put mathematics into my list labels? First, realize that the way LaTeX uses the term **label** in the context of lists is different from how much of the rest of the world uses the term in this context. In our case the `@label` attribute describes the style of the grouping markers. For example, bullets versus squares on items of an unordered list, or Roman numerals versus Arabic numerals on an ordered list. So this attribute conveys information *about* the list, not content *of* the list. And even if you tried putting an `<m>` tag into the value of the attribute, you would not have any luck since XML does not even allow that construction. Finally, there is no real good way to accomplish this in HTML, so conversion to that format would be difficult.

The alternative is to use a description list, with tag `<dl>`. You are reading one right now. Put your mathematics in the `<title>` and the associated content into the remainder of the `<li>`.

Why are theorems, definitions, examples, remarks, etc. all numbered using the same counter? The following is an argument in favor of using common counters for blocks of similar appearance. The argument is stronger in the context of using a printed copy of the book, where physical page flipping is necessary, but also applies to scrolling through a (long) page in a web browser.

Suppose your math professor gave you a note to review Theorem 2.4.7 in your textbook. The "2.4" is useful information directing you to Chapter 2, Section 4. You can tell by the "7" that the theorem is

probably not right at the beginning of the section, so you open to the middle of the section. You find yourself on a page with no theorems, but you do see Example 2.4.11. What do you do: flip forward or flip backward?

If theorems and examples are numbered using separate counters, you have no information about which way to go. You need to make a random decision, and flip pages until you find another theorem that you can use as a guidepost. And theorems may be rare and sparse, so it may take quite a bit of page flipping to find that guidepost. You may end up at Theorem 2.4.8, telling you that you need to flip backward now. But how far? Will it be one page earlier or twenty?

If theorems and examples are using the same counter, then you know that you need to flip backward. And perhaps more importantly, the oblivious reader who thinks they are looking for "2.4.7" (and is not thinking about "Theorem") sees the "2.4.11" and correctly flips backward without even realizing the potential for different counters. As they pass Definition 2.4.10 and Example 2.4.9, they have a sense of the pace at which they are converging on Theorem 2.4.7. This is the main argument for the use of common counters: it makes everything easier to locate.

For another point, perhaps you have read a math book in the past and have seen something like "and according to 2.4.7...". What if the book has Theorem 2.4.7 and also Example 2.4.7? Which one is the author talking about? If you are lucky, you are conscious that there are multiple possibilities. If you are unlucky, you flip to Example 2.4.7 but the author meant Theorem 2.4.7, and you go a bit mad trying to make Example 2.4.7 logically relevant to the reference.

MathBook XML makes it easy to avoid this particular annoyance, because cross references can identify their "type", especially through the use of the global "autonaming" feature. But even when the text explicitly says "and according to Theorem 2.4.7...", with humans being human, some readers will still focus on the "2.4.7" and begin a search for that number rather than the theorem with that number. With common counters, once 2.4.7 is a Theorem, there will never be an Example 2.4.7.

One counterargument is that it feels "wrong" to allow for an Example 2.4.9 and an Example 2.4.11 with no Example 2.4.10 existing in between. This violates our instinct to categorize and order objects. And perhaps the "missing" Example 2.4.10 will indeed cause some confusion to some readers. However, we suggest that such idealized views should be subservient to our goal of producing textbooks that provide a useful resource for the vast majority of our students.

We plan to allow figures and tables to use different counters, since they look obviously different, so you can quickly distinguish the previous, or next, item as you scan a page. Notice that it is their *distinctive appearance* that is the criteria for an independent counter. For example, numbers for displayed equations meet this criteria. They have their own counter, they are displayed distinctively when originlly formatted, and a cross-reference emphasizes their distinctive type through the use of parentheses (e.g., Equation (2.4.7)).

Also, as an author, recognize that there is a very flexible mechanism for making lists of objects that may be included in the `<backmatter>`. To continue the example here, you could make a list of all the theorems in the book, and a separate list of all the examples. Each list would be in the order of appearance, include the number (and a title if you provide one). In HTML output, each is a knowl which will quickly provide the content (independent of location), and also provides an "in-context" link to take you to the location for surrounding material. This useful feature requires very little additional effort, especially if you title your blocks as you author them.

Appendix B

(*) Text Editors

This appendix has information about using various text editors efficiently with MathBook XML source. The choice of an editor that suits you is a big part of being a productive author. Despite not being open source, we are partial to Sublime Text, due to its unlimited trial period, reasonable licensing (cost and terms), range of features, and cross-platform support. So we lead with Sublime Text, but also include Emacs and XML Copy Editor.

B.1 Sublime Text

Dave Rosoff

Sublime Text is a fast cross-platform editor with thousands of user-contributed packages implemented in its Python API. It is not free or open-source, although most of the user-contributed packages are both. Development is active as of June 2016.

Here, we outline several of the most important Sublime Text features that will help you to minimize your typing overhead and work more efficiently with your MathBook XML project. We also introduce the MBXTools package designed to help MathBook XML authors work more efficiently.

Sublime Text 2 and 3 are both available for an unlimited evaluation period, but a licence must be purchased for continued use. I have found the additional features of Sublime Text 3 to be well worth the cost of the license.

B.1.1 Settings

Sublime Text settings are stored and managed in a collection of JSON files as key-value pairs, in files that have a `.sublime-settings` extension. You change the settings by visiting these files and editing the values away from their defaults.

To edit your Sublime Text settings, you can use the Preferences/Settings — User menu (Sublime Text/Preferences... on OS X). Make sure that when you go to edit Settings, you always choose the User option. Changes to Default settings files will be overwritten when Sublime Text updates. It is recommended to use the Default files to see what settings are available to change. There are a lot, and not all are documented.

All Sublime Text users should be aware that a particular view (buffer) may receive settings in several different ways, e.g., from global default settings, from global OS-specific settings, from package-provided settings, from user-provided settings, and so on.

Key bindings are also stored in files with a similar format. There are only so many keyboard shortcuts available, although Sublime Text does support multistep shortcuts like Emacs. If you find that you wish to reassign shortcuts, this is certainly possible through the Preferences/Key Bindings — User menu (Sublime Text/Preferences... on OS X).

B.1.2 Package Control

Sublime Text's Python API exposes a lot of the Sublime Text internals to plugin and package authors. Packages extend Sublime Text's functionality, much like Emacs major modes. A package usually consists of some Python scripts that define Sublime Text events and actions, some text-based configuration files (XML/JSON/YAML files defining language syntax, symbol recognition, custom snippet insertion triggers and contexts, keybindings for new and old commands, etc.), and perhaps some other stuff too. These typically get bundled into a `.zip` archive that is disguised with the unusual extension `.sublime-package`. These archives live in the Packages directory, accessible via the Preferences menu (the Sublime Text/Preferences menu on OS X). Sublime Text monitors the `Packages` directory for changes and reloads all affected plugins on the fly.

The first thing you should do after installing Sublime Text is install the Package Control package. This package manager operates within Sublime Text to automatically fetch updates for packages you have installed (unless you disable this feature). You can also list currently installed packages, find new packages to investigate, remove packages, etc.

Thousands of user-contributed packages are available for easy installation via Package Control. It is possible to maintain packages by hand, since most package authors publish via GitHub, but Package Control is the universally recommended method of obtaining, managing, and removing packages for your installation.

1. Visit the Package Control download site.
2. Find the Sublime Text console command (make sure the correct version of Sublime Text is selected) and copy it to the clipboard.
3. Open the Sublime Text console (`Ctrl-`` `) and paste the command into the window that appears, then press `Enter`.

Having installed Package Control, you can use the command palette to deploy its commands, such as Install Package, List Packages, and Remove Package. See the documentation for more. A few packages that are especially useful are recommended throughout this section, and summarized in Subsection B.1.9.

B.1.3 (*) Keyboard Shortcuts

To be written.

B.1.4 Project Management

Like many modern editors, Sublime Text has good project management features. These allow files that are part of a larger project to work together. For example, Sublime's Goto Anything command allows quick access to any file in a project. The Find in Project command permits users to search and replace (with or without regular expressions) across an entire project. Matches are displayed in a text buffer and double-clicking opens the relevant file at the appropriate position.

The sidebar provides a convenient view of all of the files and directories in a project—or, if you like, a filtered view, where files of your choice are excluded. The MBXTools package (Subsection B.1.7) also makes some use of project-specific settings in order to provide some of its functionality.

B.1.4.1 The Open Folder Command

The easiest way to make use of the project management functionality is to store related files in a single directory and its subdirectories. If you then use the File/Open Folder... command, the entire directory is opened and all its subdirectories and files are shown in the sidebar. You can toggle the sidebar with either the command palette or directly with `Ctrl+K, Ctrl+B` (`Cmd+K, Cmd+B` on OS X).

By making use of this command you are already using project management, even if you never save your project. Sublime Text always has an implicit project open if you don't open an explicit one. This is good enough for many users a lot of the time, since it provides the most useful feature (Find/Find in Project). The Goto/Go To Symbol in Project command is also useful, but not fully implemented in MBXTools (Subsection B.1.7). Some of the benefits of explicit project management are outlined below.

B.1.4.2 Explicit Projects

To save your project explicitly, use the Project menu to choose Save As Project... and choose an appropriate name and location. For a MathBook XML project, this would probably be the same name and location as the document root file. Use the Project menu commands to open and close your project.

There are a few benefits to using an explicit project to group files.

- You can group together files and folders in different parts of the filesystem, instead of being restricted to subtrees.
- You can have project-specific settings that are different from Sublime Text's defaults and different from your user preferences (⟨⟨subsection-settings⟩⟩).
- Sublime's project workspaces will remember which files you had open when you last closed the project, and at which positions.
- If you get very fancy, you can have multiple workspaces for the same project, with different filters and views for different purposes.
- It is fine to include `.sublime-project` files in Git repositories, but `.sublime-workspace` files should ***never*** be so included (according to the Sublime Text documentation).

B.1.4.3 Using the Sidebar

The project sidebar allows you to view the entire directory tree (rooted at the folder you opened with the Open Folder command), or, if you've opened an explicit project as described above, all of its files and folders. You can use the sidebar to copy, move, rename, delete, and duplicate files, for example, as well as opening them.

The package SideBarEnhancements is highly recommended (install via Package Control). It makes the sidebar much more useful.

An alternative to the sidebar that Emacs users especially will find helpful is the dired package. The link is to a git repository since the package is no longer available from Package Control. This package allows you to browse the directory tree in a Sublime Text buffer. You can rename and move files within it—using all your favorite Sublime commands, including multiple selections (Subsection B.1.5). You might also try the SublimeFileBrowser package, which is actively maintained, available in Package Control, and seems to provide similar functionality.

B.1.5 Multiple selections

Multiple selections are the single most useful and irreplaceable feature of Sublime Text, the one that will keep you coming back. From the documentation:

> Any praise about multiple selections is an understatement.

The base functionality of multiple selections is simple. Hold down the `Ctrl` key (`Cmd` on OS X), and click somewhere in the open view to get a second cursor. Continue to add more cursors. All of them will behave together when you type: text will be inserted, most snippets or other text commands function as usual, etc. Even mouse commands work in an intuitive way with multiple selections.

It is hard to explain exactly what makes multiple selections so powerful. You just have to try it for yourself. Here is a typical example. In a structured document, many bits of text occur quite frequently—element and attribute names, for example. You may want to update several occurrences of a fragment at once—making several identical changes. Sublime's Quick Add Next command (`Ctrl+D`/`Cmd+D`) makes this a snap.

1. Place the caret somewhere in the word you'd like to modify.
2. Use Quick Add Next to expand your (empty) selection to the current word.
3. Use Quick Add Next again to add the next instance to the selection, which will then typically be disconnected.

4. Continue to Quick Add Next as many times as you like. Use Quick Skip Next (`Ctrl+K, Ctrl+D`/`Cmd+K, Cmd+D`) to jump over instances you would like to leave alone. If you go too far and select in error, hit `Ctrl+U/Cmd+U` to undo.

5. Make your modification, only one time.

Another example that occurs frequently when authoring XML is when you use the Wrap with Tag snippet (`Alt+Shift+W`/`Ctrl+Shift+W`). This snippet wraps the selection(s) in a `<p>` tag, with the tag name highlighted in both the start and end tags. If the `p` element is not what you wanted, just type. Both tags are replaced. This is a huge benefit to the XML author that makes essential use of multiple selections, even though you are barely aware of this as you use the feature.

Column selection allows you to select a rectangular area of a file. This is unbelievably useful when editing a structured document. There are lots of ways to do it (see the Sublime Text documentation for a mostly exhaustive list), but the most frequently used is to hold down `Shift` while clicking and dragging with the right mouse button (on OS X, hold down `Option` while dragging with the right mouse button). See the documentation for keyboard-based shortcuts.

Column selection becomes even more useful when used in combination with the keyboard shortcuts for moving and selecting, such as `Ctrl+Shift+Right` (select to end of word) and `Shift+End` (select to end of line).

Yet another example of the appallingly great utility of multiple selection comes when copying and pasting from a different file format. Suppose you have copied some lines of text and wish each such line to become a list item in your MathBook XML source.

1. Use column selection, as described above, to select each line individually.

2. Use Wrap with Tag to wrap each of the selected lines with matched begin/end `<li>` tags, all at once.

3. Now you have to select the lines again, to wrap them with matched begin/end `<p>` tags. First, hit `Shift+End` to select to end of line.

4. If your lines are wrapped, you may need to hit `Shift+End` again to get to the end of the wrapped lines.

5. Now you've selected too far: the `</li>` are selected as well. Hold down `Ctrl+Shift` and hit the left arrow twice (unselect by word). (After a little practice, steps like this seem automatic.)

6. Use Wrap with Tag to wrap each of the selected lines with matched begin/end `<p>` tags, all at once.

This does take a little mouse-work, but the keystroke savings can be considerable. (The Emmet package, described in Subsection B.1.6, provides an even quicker way to do this task and much more complicated ones.)

There are so many incredibly handy ways to use multiple selections that we will forgo any further examples to leave the reader the pleasure of discovering her own favorites. One particularly helpful package is Text Pastry, which provides some autonumbering and text insertion commands that work nicely with multiple selections. There are also a handful of packages that extend multiple selection functionality, such as PowerCursors and MultiEditUtils. PowerCursors allows you to add cursors and manipulate them without using the mouse. MultiEditUtils provides additional text processing commands designed to work with multiple selections.

B.1.6 Emmet

Emmet is the most downloaded plugin for Sublime Text (1.82 million installs via Package Control). It is mostly used by HTML and CSS authors and provides a lot of functionality for them. It is also useful for writing XML, as we see below. The main benefits of working with Emmet are ease of tag creation, manipulation, and removal.

Emmet by default overrides Sublime's binding for the `Tab` key, endowing it with new behavior (the command Expand Abbreviation). This new behavior is to create a matching XML tag pair for whatever word is to the left of the caret, or with whatever words are selected. For example, if you were to type "ol" and press the `Tab` key, the resulting text would be

```
<ol></ol>
```

with the caret positioned between the two newly created tags. Pressing `Tab` a further time moves the caret to the right of the end tag.

Emmet will produce any word it does not recognize into a matched tag pair when the Expand Abbreviation command is run. Some XML elements are empty, though. Within a matched tag pair, the command Split/Join Tag (`Ctrl+Shift+`/Cmd+Shift+``) will contract it into an empty tag, removing any text between the existing begin and end tags. (If the caret is *inside* a tag for an empty element, this command replaces the empty element with a matching begin/end tag pair.)

The default behavior (creating tag pairs whenever `Tab` is pressed) interferes with Sublime Text's usual Tab-completion, which may be undesirable. It may be disabled by setting

```
"disabled_keymap_actions": "expand_abbreviation_by_tab"
```

in the Preferences/Package Settings/Emmet/Settings — User file. The functionality of Expand Abbreviation will still be available through `Ctrl+E`.

For a more involved example of abbreviations, suppose you have pasted the items of an ordered list. Now you need to structure it with `ol`, `li`, and so on.

```
Lists are often good.
You can provide list items with <c>@xml:id</c>.
You probably don't want to number them, though.
```

The desired output is:

```
<ol>
    <li xml:id="item1">Lists are often good.</li>
    <li xml:id="item2">You can provide list items with <c>@xml:id</c>.</li>
    <li xml:id="item3">You probably don't want to number them, though.</li>
</ol>
```

Using Emmet, one produces it by executing the Wrap as you Type command (`Ctrl+Shift+G`/`Ctrl+W`) and entering the following expression in the minibuffer.

```
ol>li[xml:id=item$]*>p
```

The `>` symbol denotes a child element, the square brackets (with or without assignment) denote an attribute list, the `$` provides the line-based numbering, and the `*` specifies wrapping each selected line with the indicated subtree (so each line is wrapped with `<li><p>`, instead of the entire selection).

Emmet can produce a large hierarchy of nested XML tags at various levels using this abbreviation syntax. For example, suppose you know that you will need to produce a tag structure of the following form.

```
<section xml:id="">
    <introduction>
        <p></p>
    </introduction>
    <subsection xml:id="">
        <p></p>
        <p></p>
        <figure></figure>
        <p></p>
        <ol>
            <li></li>
            <li></li>
            <li></li>
        </ol>
    </subsection>
    <conclusion>
```

```
        <p></p>
    </conclusion>
</section>
```

Admittedly, this is a bit much, but it makes the point. The Emmet "abbreviation" for this structure is:

```
section[xml:id]>introduction>p^(subsection[xml:id]>p*2+figure+p+ol>li*3)^^conclusion>p
```

Upon typing this string and placing the caret to the right of it, hit `Ctrl+E` (or `Tab`, if you didn't disable the Emmet default). The entire tree structure is created immediately, with tab stops for the missing attribute values and for each matching begin/end pair.

The Expand Abbreviation As You Type command allows you to tweak such abbreviations interactively. Hit `Ctrl+Alt+Enter` and type the expression above into the minibuffer at the bottom of the window, watching the tree appear as you type.

Emmet is a very powerful package that can do much more than is outlined here. However, it is by default mostly adapted to writing CSS and HTML. Customizing it to work more directly with MathBook XML is an ongoing project. You can discover more about Emmet by examining the Emmet documentation or poking around in the Settings and Keymap files.

B.1.7 MBXTools—a Sublime Text package for MathBook XML

MBXTools is a Sublime Text package designed to assist authors using MathBook XML. It is very experimental and may behave unexpectedly.

The package owes its inspiration and much of its code to the excellent LaTeXTools package. Please let the author know of any bugs you find or any features you would like to see included in MBXTools by creating a GitHub issue.

B.1.7.1 Installation

via Package Control It is recommended to install MBXTools via Package Control. If you have not installed Package Control yet, you should do that first (and restart Sublime Text afterward).

After Package Control is installed, use the `Install Package` command to search for the MBXTools package, and select it from the Quick Panel to install. This method of installation allows Package Control to automatically update your installation and show you appropriate release notes.

via git You may also install MBXTools via `git`. Change directories into your `Packages` folder. To find the `Packages` folder, select Browse Packages from the Preferences menu (from the Sublime Text 3 menu on OS X). Make sure you are in the `Packages` folder and *not* `Packages/User`.

Then, run

```
git clone https://github.com/daverosoff/MBXTools.git
```

and restart Sublime Text (probably not necessary).

B.1.7.2 Usage

You can activate the package features by enabling the MathBook XML syntax. The syntax definition looks for `.mbx` file extensions, which most of us do not use (yet?). If your MathBook XML files end with `.xml`, you will either need to add a comment to the first line of each file (after the XML declaration):

```
<!-- MBX -->
```

or you will need to enable the syntax manually using the command palette. To enable it manually, open a MathBook XML file and press `Ctrl+Shift+P` (`Cmd+Shift+P` on OS X) and type `mbx`. Select "Set Syntax: MathBook XML" from the list of options.

You should see the text "MathBook XML" in the lower right corner if you have the status bar visible (command palette: Toggle Status Bar).

There are only a few features implemented so far.

1. If you have some sectioning in your MBX file, hit `Ctrl+R` (`Cmd+R` on OS X) to run the Go To Symbol command. You should see a panel showing all the subdivisions' `@xml:id` names.

2. If you have been using `@xml:id` to label your stuff, try typing `<xref ref="` (the beginning of a cross-reference). Sublime Text should show you a panel containing all `@xml:id` values along with the elements they go with. Choose one to insert it at the caret and close the `xref` tag. Alternatively, type `ref` and hit `Tab` to activate the `xref` snippet. Then hit `Ctrl+l` followed by `x` or `Ctrl+l` followed by `Ctrl+Space` to bring up the completions menu. There are several variants of the `ref` snippet, namely `refa`, `refp`, and `refpa`.

3. Type `chp`, `sec`, `ssec`, or `sssec` and hit `Tab` to activate the subdivision snippets. A blank `title` element is provided and the cursor positioned within it. As you type, the `@xml:id` field for the subdivision is filled with similar text mirroring the title you are entering.

B.1.7.3 Known issues

1. When manually adding an `xref` (not using the snippets or autocomplete), you will frequently see a spurious "Unrecognized format" error.

2. The `ref` snippet does not bring up the quick panel. Should it?

3. Recursive search through included files for labels is not yet implemented.

   This will only work for `xref` completion, not Go To Symbol.

4. Nothing has been tested on OS X or Linux.

B.1.8 (*) Sublime Linter

To be written.

B.1.9 Recommended Packages

1. Package Control

2. Emmet

3. SideBarEnhancements

4. PowerCursors

5. MultiEditUtils

6. Text Pastry

7. Git or SublimeGit

8. SublimeLinter

9. MBXTools

B.2 emacs

Jason Underdown reports on 2016-05-12 that emacs' nXML mode (`www.gnu.org/software/emacs/manual/html_mono/nxml-mode.`
works well with a RELAX-NG schema. While we work on building a hand-crafted RELAX-NG schema, you can use the trang (`www.thaiopensource.com/relaxng/trang.html`) tool to convert the MathBook XML DTD to a RELAX-NG schema.

> You simply put your cursor at any point in the document, start a new tag with < and then call the `completion-at-point` function (I bound it to the key-chord: `C-<return>`) to get a list of possible completions. Or you can start typing a few characters to narrow the list of possibilities. It will also let you know if the element you are trying to insert is invalid.
>
> —Jason Underdown

B.3 XML Copy Editor

Michael Doob reports on 2017-02-03 that XML Copy Editor (`xml-copy-editor.sourceforge.net`) works well, in particular on Windows. This is an open source program, for Windows and a variety of popular Linux distributions, that supports both DTD and RELAX-NG schemas. It is less of a general programmer's editor and more like dedicated tools for working strictly with XML documents.

B.4 (*) vi, vim

Contributions welcome.

Appendix C

Revision Control: git

Authoring a textbook without revision control is like driving without a seat belt. Sooner or later, you will wish you had used it. git is a popular program for revision control for software projects, and works quite well with PreTEXt, though not perfectly. Notes here are designed to help. For more on git itself, in the context of authoring a book, see Git for Authors, by Robert Beezer and David Farmer at `mathbook.pugetsound.edu/gfa/html`.

Word Wrap git is designed for code, where a newline often expresses the end of a statement. In PreTEXt, it might make sense to author an entire (long) paragraph without any newlines. If so, a line-oriented file diff is not so useful. Fortunately, git has a flag, `--word-diff`, which does an excellent job of displaying small edits precisely.

Messages for Commits and Merges When you make a commit or merge, you can supply a message at the command line with the `-m` argument. Otherwise you get thrown into an editor, with the default being `vi`, which can be hard to get out of if you have not used it before. Better, as Joe Fields suggests, is to tell git which editor you want to use. To set `pico` as the default editor, the one-time command-line incantation would be:

```
git config  --global core.editor "pico"
```

You can also directly edit the configuration file at `~/.gitconfig`. More suggestions can be found on this thread on StackOverflow at `stackoverflow.com/questions/2596805`.

Appendix D

Windows Installation Notes

Dave Rosoff

This appendix explains the best known way to install the MathBook XML toolchain in a Windows environment, rather than a Unix-flavored operating system (such as Linux and Apple's OS X). It has been tested on Windows 7, 8, and 10. We assume that none of the listed tools or equivalents have been previously installed. That may complicate matters. This is especially true if you use Cygwin, or if you have already installed Python on your machine. MathBook XML compatibility with existing Python installations is addressed elsewhere in this document (⟨⟨python-mbx-compatibility⟩⟩).

If you have Windows 10, be sure to read about WSL in Appendix E, which could be a whole lot easier to setup and maintain.

D.1 Setup

In this section, we do some initial setup, establish notation, and issue warnings. Some of the steps in this process are dangerous. Typos could lead to an unstable system, or possibly even to unrecoverable system errors. Double-check everything.

D.1.1 Notation

Strings enclosed in `<angle brackets>` are variables whose values you should substitute in typed expressions. `<username>`, for example, should be replaced with your Windows username (e.g., mine is `drosoff`). Throughout this installation process it is very important to pay attention to the direction of slashes `/` and backslashes `\`.

D.1.2 Initial Windows setup

It is easier to see what is happening if your Windows file browser is not set up to hide file extensions from you. Disable the "hide file extensions" behavior before proceeding. In Windows 7/8, this can be done through the Control Panel. In Windows 10, there is a checkbox somewhere in the ribbon for it.

List D.1.1.

- On Windows 7 or 8:
    1. Open the Start Menu and type "Control Panel". Select the `Control Panel` entry from the popup list.
    2. Type "Folder Options" into the search box in the Control Panel window. Select `Folder Options` when it appears.
    3. Select the View tab.

4. Uncheck the box for "Hide extensions for known file types".
5. Click OK until there are no more OKs to click.

- On Windows 10:

1. Open the Start Menu (icon shaped like a Window at the bottom left, typically). Select the `File Explorer` option, right above `Settings`, from the popup list.
2. Click on this, and then select `View` from the "ribbon lists" of options along the top.
3. After clicking this, on the right there should be a check box for "File Name Extensions". Click this box; that should do it.

D.1.3 A word on path names

An appallingly large fraction of the difficulties of using GNU/Linux-based utilities with Windows come from the differences in formatting path names. Windows path names begin with a drive letter (usually "C") and a colon. Like all path names, they describe a path in a rooted tree. The root directory (folder) in Windows is called `\`, a backslash. Note the direction carefully. Children of the root node are either subdirectories or files in the root directory (leaves). The path to my downloads folder is:

```
C:\Users\drosoff\Downloads\
```

The trailing backslash is often unnecessary, but it is an easy way to see immediately whether a path name refers to a file or to a directory. Windows path names are not case-sensitive.

Linux/Mac OS X path names are quite similar, but lack drive letters, start with an explicit reference to the root, use forward slashes, and are case-sensitive (more or less so, in Mac's case). A path to a typical Linux user's download folder might be

```
/home/typical.username/Downloads
```

Again, Linux pathnames are case-sensitive and Mac OS X pathnames are typically 'case-preserving'. The Git Bash shell for Windows is an emulation of a Linux environment, and the utilities within it expect path names that follow Linux conventions. So we conform to this expectation as follows.

1. Remove the colon, but keep the drive letter.
2. All backslashes `\` become slashes `/`.
3. Add an initial slash preceding the drive letter.

The path name to my Windows download folder becomes

```
/c/users/drosoff/Downloads
```

Even though Git Bash is pretending to be a Linux shell, path names are still the underlying Windows path names, and therefore are not case sensitive. You can verify this using tab-completion.

Path names that begin with the drive letter (Windows) or the root `/` (Linux/Mac OS X) are called **absolute path names**. Their referents do not depend on the location from which the path name is invoked. **Relative path names**, on the other hand, begin in the so-called **current working directory**. A relative pathname might look something like this:

```
../../examples/sample-article/sample-article.xml
```

The symbol `..` is a shortcut for the parent of the current directory. Thus, the relative path name above means from where we are, ascend two levels, then descend into the `examples` and `sample-article` subdirectories, and find the file `sample-article.xml`.

Path names that contain spaces are *evil*, and should be avoided in many cases. Unfortunately, all Windows default program installation locations contain at least one space (directory name "Program Files"). This does not appear to cause problems, except when installing ImageMagick (Section D.5). To be extra careful, you could always choose an installation location that is free of space characters.

D.1.4 Do I have 64-bit Windows?

Find out on Windows 7:

1. Open the start menu and type "Computer". Right-click the `Computer` item in the popup menu.
2. Select `Properties` from the drop-down menu.
3. Read in the right-hand side of the pane to find the "System" heading.
4. From the "System type" entry, read whether you have a 32- or a 64-bit OS.

Now you are ready to begin installing the various pieces of the MathBook XML puzzle. The first one, `xsltproc` (Section D.2), is the most annoying. You might want to go get a snack or another cup of coffee.

D.2 Installing xsltproc

D.2.1 xsltproc binaries

This is the most annoying part of the installation. Obtain four zip archives from Igor Zlatkovic's FTP site that hosts the most recent Libxml binaries for Windows. At the time of this writing, the 64-bit binaries were considered experimental. I have had no trouble using the 32-bit binaries on my 64-bit Windows 7 system, so I suggest that all MBX users download the most recent 32-bit version of the following libraries:

List D.2.1.

1. `iconv` (filename something like `iconv-1.9.2.win32.zip`)
2. `libxml2` (filename something like `libxml2-2.7.8.win32.zip`)
3. `libxslt` (filename something like `libxslt-1.1.26.win32.zip`)
4. `zlib` (filename something like `zlib-1.2.5.win32.zip`)

We only need a handful of files from these archives. So the simplest thing is to leave them in your Downloads folder and grab what we need. Create a new folder `C:\xsltproc` (it can be anywhere, as long as it's a new location). We'll call this location `<xsltproc>` in case you named your folder something different.

Extract the following files from the four zip archives you downloaded above into `<xsltproc>`:

List D.2.2.

1. From `iconv-*.win32.zip`:
   (a) `iconv-*.win32\bin\iconv.dll`
2. From `libxml2-*.win32.zip`:
   (a) `libxml2-*.win32\bin\libxml2.dll`
   (b) `libxml2-*.win32\bin\xmllint.exe`
3. From `libxslt-*.win32.zip`:
   (a) `libxslt-*.win32\bin\libexslt.dll`
   (b) `libxslt-*.win32\bin\libxslt.dll`
   (c) `libxslt-*.win32\bin\xsltproc.exe`
4. From `zlib-*.win32.zip`:
   (a) `zlib-*.win32\bin\zlib1.dll`

D.2.2 Change PATH environment variable

Note: if you prefer not to meddle with this, it can be avoided. Now, we need to make sure your system can find these files when we need them.

List D.2.3.

1. Open the Start menu and start typing "Edit the system environment variables". Select this option when it becomes visible.
2. Click the Environment Variables button near the bottom of the dialog.
3. In the bottom part of the dialog labeled "System environment variables", look for a variable named `PATH`. You may need to scroll.
   (a) If you do not find one, create it using the New... button. Make sure to use all capital letters. (This really shouldn't happen. Make sure you are editing the *system* environment variables, not the *user* environment variables.)
   (b) If you do find the `PATH` variable, select it and click the Edit... button.
4. Regardless of which of steps 1 and 2 you followed, now you should see a dialog with two text fields. Your variable name should be `PATH`.
5. If you created this variable, populate the second field with the full path name of `<xsltproc>`, the location where you put the seven files from Igor's zip archives. For me this looks like `C:\xsltproc`.
6. If you are editing the `PATH` variable, place the cursor in the existing value and press the End key, so that the cursor moves to the back of the line. The `PATH` string is a `;`-delimited list of full path names, so append the string `<xsltproc>;` (note the semicolon) to the existing value. If you named <xsltproc> as we suggested above, then the last part of your `PATH` variable is now `C:\xsltproc;`.
7. Click OK to save changes.

Congratulations, you have successfully installed `xsltproc`.

Note that you have installed the `xmllint` utility as part of this procedure. This utility will allow some text editors to **lint** your MathBook XML files, that is, to automatically detect and highlight errors, and perhaps even to explain them.

D.3 Installing git

In this section we install the `git` version control system and some tools to interact with it, including a fairly full-featured emulation of the `bash` command line shell. I strongly recommend you use the Git Bash shell or another `bash` emulation, so that you can use Linux commands referenced elsewhere.

One feature in particular, `pdfcrop`, has not been made to work in the normal Windows `cmd` shell, although the rest of MathBook XML has. To generate images using the `mbx` script Section 8.3, you will need the Git Bash shell or something like it.

D.3.1 Steps to install git

List D.3.1.

1. Visit the official `git` download page (download starts automatically) and obtain the latest binary for your system.
2. Find the installer in your Download location and run it.
3. Choose whatever location you like for the `git` installation folder. I recommend you use the default.

4. At the "Adjusting your PATH environment" dialog, select either of the first two items. I recommend the second, "Add git executable to Windows PATH". This will allow you to use `git` from within other Windows programs, such as Sublime Text or other text editors, which can be extremely convenient. If you are apprehensive about adding `git` to the Windows PATH, select the first option. I do not recommend the third option.

5. Accept the default options for all the remaining prompts.

D.3.2 Changing the path with .bashrc

In Subsection D.2.2, we promised that you could avoid messing with the Windows environment variables. If you install something else later that wants to use `xsltproc`, then this might not be the best idea. But if you are only going to use it from within Git Bash, then this will work fine.

From the Git Bash command prompt, enter this line of text and hit Enter. Do not make any typos. You should substitute your value of `<xsltproc>` where indicated, but make sure to conform to the conventions at the end of Subsection D.1.3 regarding Windows path names in Git Bash. (I warned you this was going to be annoying.)

```
echo "export PATH=<xsltproc>:$PATH" >> ~/.bashrc
```

You may get a message from Git Bash the next time you run it about `.bash_profile`, which you may safely ignore.

Congratulations, you have successfully installed `git`.

D.4 Installing Anaconda

Anaconda is a well-regulated development environment for Python under Windows, and I recommend it for users who do not already have Python installed. The essential `mbx` script has recently been updated to support both Python 2 and Python 3. Therefore we make no recommendation about which Python version to choose.

If you already have a working Python installation, skip to Section D.5.

You have some choice with your Anaconda installation. It actually supports the installation of several independent Pythons side-by-side (since both 2.7 and 3.x are in active use, this is more reasonable than it seems). If you do not need Python for anything else or are simply a minimalist, Miniconda is also an option. Miniconda installs no packages, but these can be installed via the `conda` utility at the command line later.

1. Download either Miniconda, Python 2.7, or Python 3.5 from the Anaconda download page.

2. Run the installer and accept all the default suggestions.

Congratulations, you have successfully installed Python.

If you do not care about images, you can stop here. Much of the MBX functionality is already present. However, to use the `mbx` script to create SVG images from sources like PDF/PNG images, Sage, Asymptote, or TikZ, you need to install ImageMagick and Ghostscript using the directions in Section D.5 and Section D.6.

D.5 Installing ImageMagick

Visit the ImageMagick downloads page and grab a binary. If you have a 64-bit Windows installation (Subsection D.1.4), use the recommended version. If you have a 32-bit installation, find the version whose filename is obtained from that of the recommended version by substituting `x86` for `x64`. For example, if the recommended version's filename is:

```
ImageMagick-7.0.1-6-Q16-x64-dll.exe
```

then a good choice for a 32-bit Windows would be

`ImageMagick-7.0.1-6-Q16-x86-dll.exe`

List D.5.1.

1. Run the installer from your download location.
2. Accept the license agreement.
3. Choose a default installation location that has no spaces in its folder name. The default choice "Program Files" causes problems because of path name issues. I chose

   `c:\ImageMagick-7.0.1-Q16\`

   It matters because the ImageMagick utility `convert` is used by the `mbx` script to convert your images into different formats. The `mbx` script will have a lot of trouble with path names that contain spaces.
4. When confronted with "Select additional tasks", make sure that the boxes for "Add application directory to your system path" and "Install legacy utilities" are checked.
5. If you like, carry out the procedure to verify your installation.

Congratulations, you have successfully installed ImageMagick.

D.6 Installing Ghostscript

Visit the Ghostscript download area and download the most current binary for either 64-bit or 32-bit Windows (Subsection D.1.4). Run the installer. You can accept almost all the default options. As with ImageMagick (Section D.5), it is probably best to choose an installation location whose path name is free of spaces, such as `c:\gs`. We refer to this installation location as `<gs>` below.

D.6.1 Change PATH environment variable

The `pdfcrop` utility needs to be told which Ghostscript command to use. We need to add the file `gswin64c.exe` to the Windows PATH. This is similar to what is done above, in Subsection D.2.2.

List D.6.1.

1. Open the Start menu and start typing "Edit the system environment variables". Select this option when it becomes visible.
2. Click the Environment Variables button near the bottom of the dialog.
3. In the bottom part of the dialog labeled "System environment variables", look for a variable named `PATH`. You may need to scroll.
4. If you do find the `PATH` variable, select it and click the Edit... button.
5. You should see a dialog with two text fields. Your variable name should be `PATH`.
6. Place the cursor in the existing value and press the End key, so that the cursor moves to the back of the line. The `PATH` string is a `;`-delimited list of full path names, so append the string `<gs>;` (note the semicolon) to the existing value. If you named <gs> as we suggested above, then the last part of your `PATH` variable is now `C:\gs;`.
7. Click OK to save changes.

Congratulations, you have successfully installed Ghostscript.

D.7 Installing pdf2svg

The installation procedure uses `git`. Open Git Bash and change to your root directory:

```
cd /c
```

Clone the repository into `C:\pdf2svg`:

```
git clone https://github.com/jalios/pdf2svg-windows.git pdf2svg
```

D.7.1 Change PATH environment variable

We need to add the `pdf2svg` program to the Windows PATH. This is similar to what is done above, in Subsection D.2.2 and Subsection D.6.1.

List D.7.1.

1. Open the Start menu and start typing "Edit the system environment variables". Select this option when it becomes visible.
2. Click the Environment Variables button near the bottom of the dialog.
3. In the bottom part of the dialog labeled "System environment variables", look for a variable named `PATH`. You may need to scroll.
4. If you do find the `PATH` variable, select it and click the Edit... button.
5. You should see a dialog with two text fields. Your variable name should be `PATH`.
6. Place the cursor in the existing value and press the End key, so that the cursor moves to the back of the line. The `PATH` string is a `;`-delimited list of full path names, so append the string `C:\pdf2svg\dist-64bits;` or `C:\pdf2svg\dist-64bits;` (note the semicolon) to the existing value.
7. Click OK to save changes.

Congratulations, you have successfully installed `pdf2svg`.

D.8 What's Missing

Development of a Windows-compatible `mbx` script (Chapter 8) is mostly complete. If you need help with `mbx`, contact Dave Rosoff. There are still a few use cases that haven't been tested, mostly those to do with Asymptote.

At present, it only seems to be possible to install Sage on Windows by way of the Windows Subsystem for Linux, available only in Windows 10.

If you find any problems or bugs, please let us know at the MathBook XML Support group in Google Groups, or email `drosoff` AT `collegeofidaho` DOT `edu`.

Appendix E

Windows Subsystem for Linux

With Windows 10 you can install the **Windows Subsystem for Linux** (WSL). This is basically Ubuntu Linux (one of the most popular versions of Linux) integrated into Windows 10 in a way that command-line Linux programs can be executed easily. News and announcements can be fouund at https://msdn.microsoft.com/commandline/wsl/about (msdn.microsoft.com/commandline/wsl/about). Michael Doob reports on 2017-06-02 that this works quite well for the programs necessary to author with PreTEXt, and provides the following help.

Installing WSL If you have the "Anniversary Edition" of Windows 10 (later than August 2016), then installing WSL is not difficult. Just follow the (reasonably straightforward) instructions given by Microsoft at the address https://msdn.microsoft.com/en-us/commandline/wsl/install_guide.

Upon completion of the installation, you should

- be able to use the `bash` command from the `PowerShell` window,
- have your own WSL userid (distinct from Windows),
- have your own WSL password (distinct from Windows).

A little background about using about the command line

- You type in commands (terminated by the Enter key) and the operating system responds. For example, if you type in `date`, the operating system responds with (what it considers to be) the date. Using the command line is an ongoing conversation between you and the operating system.
- The `sudo` command: when a command starts with `sudo`, the rest of the command is executed with administrative privileges. This is needed, for example, to install software or update the operating system. You must give your password when you run `sudo` (although you get a little window of time after the first usage when it is not necessary to do so).
- The `sudo apt-get update` command: this is used to resynchronize the local listing of installed packages with those in the official repository.
- The `sudo apt-get upgrade` command: this is used to bring all the local software up to date with those in the official repository.

Run `sudo apt-get update` followed by `sudo apt-get upgrade` with a new system to bring it up to date. It is a good idea to repeat this frequently to have the latest software on your computer.

Installing software The default configuration of WSL does not have the software needed for creating documents with PreTEXt. There are a few commands to be run before you can get started.

- The program **git** is used to download the PreTEXt software onto your local computer. It is installed with the command `sudo apt-get install git`.
- The program **xsltproc** is used to create your readable documents. It is installed with the command `sudo apt-get install xsltproc`.

After these are installed, you are ready to set up PreTEXt.

Putting PreTEXt on your computer Here are the steps necessary to get the PreTEXt software onto your computer:

- Make a new directory `mkdir mathbook`
- Make your own clone of the PreTEXt repository `git clone https://github.com/rbeezer/mathbook.git`
- Move to the new directory `cd mathbook`
- Initialize the new directory with `git pull`

This last command synchronizes your files with those in the official repository. You should run it frequently to keep your files up to date.

The simplest example Here is a brief description of the use of WSL to create readable files. You, as the author, create the XML file. The system will contain an appropriate XSL file that translates your XML file to something readable.

Several editors come with WSL by default including NANO, PICO, VI, and VIM. Here are the steps to follow:

1. Type the command `cd` to align yourself in your home directory.
2. Use one of the editors to create a file called `hw.xml` (you could use the command `nano hw.xml`), and add the following text:

```
<?xml version="1.0" encoding="UTF-8" ?>
<mathbook>
    <article xml:id="hw">
    <p>Hello, World!</p>
    </article>
</mathbook>
```

3. Run the command `xsltproc mathbook/xsl/mathbook-html.xsl hw.xml` Upon completion, your should have a file called `hw.html`.
4. Now the tricky part: you want to view the `hw.html` file in a browser, but the usual Windows programs cannot see the files created within WSL. So we have to copy them to a place where they are visible. Fortunately, this is pretty easy to do. To put `hw.html` on your desktop, use the command `cp hw.html /mnt/c/Users/username/Desktop` (note that "username" must be replaced by your Windows user name). Once the file is on the desktop, a double click will open it in a browser.

The `edit-xsltproc-view` cycle just given may seem daunting at first blush. Some things that can help:

- Pressing the up arrow when at the command line displays the previously executed commands. Hitting the enter key while such a command is displayed executes it. This saves a lot of retyping.
- It is possible to define aliases to shorten commands. Your local Linux guru can show how this is done.
- It is possible to define scripts to shorten multiple commands. Your local Linux guru can show how this is done.

Appendix F

GNU Free Documentation License

0. PREAMBLE The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque

copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties — for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers

that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or

in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  YEAR  YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with... Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.